Proved Implementations of Cryptographic Protocols in the Computational Model

David Cadé

UNIVERSITÉ PARIS.
DIDEROT (Paris 7)
SORBONNE PARIS CITÉ

ÉCOLE DOCTORALE : Sciences mathématiques de Paris Centre — ED 386
Laboratoire ou unité de recherche : PROSECCO INRIA

DOCTORAT

Discipline : Informatique

CADÉ DAVID

IMPLÉMENTATIONS DE PROTOCOLES CRYPTOGRAPHIQUES PROUVÉES DANS LE MODÈLE CALCULATOIRE
PROVED IMPLEMENTATIONS OF CRYPTOGRAPHIC PROTOCOLS IN THE COMPUTATIONAL MODEL

Thèse dirigée par BLANCHET Bruno

Soutenue le 16 décembre 2013

JURY

M. Blanchet Bruno	, Directeur de thèse
Mme. Cortier Véronique	, Rapporteur
M. Barthe Gilles	, Rapporteur
M. Goubault-Larrecq Jean	, Examinateur
M. Küsters Ralf	, Examinateur
M. Treinen Ralf	, Examinateur
M. Fournet Cédric	, Examinateur

Remerciements

Tout d'abord, je voudrais remercier mon directeur de thèse, Bruno Blanchet, pour ses encouragements et son aide continue pendant ces quatre dernières années. Sans lui, je n'aurais pas pu finir ce travail.

Ensuite, je tiens à remercier mes rapporteurs qui ont accepté de lire ce manuscrit, ainsi que tous les membres du jury.

J'ai été dans trois équipes différentes pendant ma thèse. L'équipe d'interprétation abstraite de l'École Normale Supérieure m'a hébergé pendant la première année de ma thèse. L'atmosphère y était très conviviale et accueillante, et j'en remercie les membres. J'étais ensuite pendant deux ans dans l'équipe de cryptographie de l'Éns. Je remercie les membres de cette équipe pour leur accueil. Je remercie enfin les membres de l'équipe Prosecco qui m'a accueillie pendant cette dernière année de thèse.

Je remercie les nombreux amis qui m'ont soutenu pendant ces quatre dernières années, dont ceux d'Alsace, où nous avions de mémorables choucroute-parties, ceux d'Alsace, qui essaient trop dur de faire des cadeaux surprise, ceux du cours de japonais, du club Animescens, et du club DDR.

Et enfin, je tiens à remercier les membres de ma famille pour leur soutien.

Table des matières

Introduction

If you do not read French, you can proceed directly to the introduction in English, page 21.

La cryptographie est actuellement utilisée dans beaucoup de contextes pour pouvoir envoyer des messages à son interlocuteur de manière sûre. Lorsqu'on paie avec une carte bancaire chez un marchand, des données chiffrées sont transmises entre la carte, le marchand et la banque. Ces transmissions permettent de s'assurer que la carte n'est pas frauduleuse, et permettent à la banque d'effectuer la transaction demandée. La façon dont les messages sont transmis entre les différents participants est ce que l'on appelle un protocole cryptographique.

La sécurité d'un protocole comme celui-ci dépend de plusieurs propriétés, comme

- la confidentialité : aucun tiers qui observerait la communication ne peut déduire le contenu des messages ;
- l'authentification : les messages du protocole qu'un participant reçoit ont bien été envoyés par son interlocuteur, et non pas par quelqu'un qui se fait passer pour lui ;
- l'intégrité : nous sommes certains que les messages que nous recevons sont exactement les messages qui nous ont été envoyés.

Primitives cryptographiques

Un protocole utilise des primitives cryptographiques pour assurer sa sécurité. Pour nous faire une idée de ce que la cryptographie peut nous permettre de faire, et pour comprendre les concepts utilisés dans la suite, présentons quelques primitives qui sont utilisées dans ces protocoles.

Pour s'assurer de la confidentialité d'un message, la plupart des protocoles utilisent du chiffrement. Le chiffrement symétrique est une manière de rendre un message illisible pour quelqu'un qui ne possède pas la clé qui a été utilisée pour chiffrer le message. Le chiffrement symétrique est composé de trois algorithmes : le générateur de clés, qui prend un paramètre de sécurité (par exemple la taille de la clé à générer) et génère une nouvelle clé k ; le chiffrement, qui prend un message m et une clé k en entrée, et qui retourne le chiffré $\{m\}_k$; et le déchiffrement, qui prend un chiffré $\{m\}_k$ et la clé k qui a été utilisée pour chiffrer le message, et renvoie le message clair m. Le chiffrement symétrique requiert des participants de connaître un secret commun, la clé, avant de pouvoir échanger des messages chiffrés.

Le chiffrement asymétrique, ou chiffrement à clé publique, permet de chiffrer un message pour un interlocuteur sans pour autant connaître un secret commun avec celui-ci. De manière similaire au chiffrement symétrique, le chiffrement asymétrique est composé de trois algorithmes. Le générateur de clés génère un nouveau couple de clés (pk, sk). La clé pk est dite clé publique et sk clé privée ou clé secrète. La clé publique sert uniquement à chiffrer et la clé privée sert uniquement à déchiffrer. Le chiffrement prend un message m et une clé publique pk, et renvoie un chiffré $\{m\}_{pk}$. Ce chiffré dépend d'une graine aléatoire : un même message chiffré plusieurs fois sous la même clé produit des chiffrés différents. Le déchiffrement prend un chiffré $\{m\}_{pk}$ et la clé privée sk correspondant à la clé publique pk, et renvoie le message clair m. La clé publique est faite pour être publiée afin que nos potentiels interlocuteurs puissent nous parler, mais la clé privée doit être gardée secrète : si un tiers met la main sur cette clé, il sera capable de déchiffrer tous les messages qui nous ont été adressés.

La signature est utilisée pour assurer l'authentification et l'intégrité. Elle est aussi composée de trois algorithmes. Le générateur de clés génère deux clés (pk, sk), sk est la clé de signature et pk est la clé de vérification qui sert à vérifier les signatures. L'algorithme de signature prend un message m et une clé de signature sk, et retourne la signature $\{m\}_{sk}$ correspondant au message m. L'algorithme de vérification prend un message m, une signature s et une clé de vérification pk. Si la signature s est bien une signature du message m sous la clé de signature sk correspondant à pk, l'algorithme répond que la signature est correcte, et répond que la signature est incorrecte sinon. De manière similaire au chiffrement asymétrique, la clé de signature sk est privée et la clé de vérification pk est à publier à toutes les personnes susceptibles de vouloir vérifier notre signature.

Les fonctions de hachage sont des fonctions qui prennent un message m en entrée et qui renvoient une chaîne de bits d'une taille fixée. Cette chaîne est appelée le haché du message m. Les fonctions de hachage sont des fonctions à sens unique : il est facile d'obtenir le haché d'un message, il suffit d'appliquer la fonction ; mais il est difficile, pour un haché h donné, de trouver un message m tel que le haché du message m soit h.

Les codes d'authentification de message (MAC : Message Authentication Code) sont des fonctions qui prennent un message m et une clé k, et qui renvoient un haché qui correspond à ces deux éléments. Le but de cette primitive est d'assurer l'intégrité des messages : la clé k permet, en quelque sorte, de choisir la fonction de hachage, qui ne sera connue que des personnes connaissant la clé. Un tiers ne pourra pas construire le MAC correspondant au message m s'il ne connaît pas aussi la clé k.

On définit la sécurité de ces primitives en fonction d'un jeu joué contre un challenger et un attaquant, qui est une machine de Turing polynomiale probabiliste. Le jeu dépend de la primitive et des propriétés de sécurité que nous voulons. Par exemple, le chiffrement asymétrique est IND-CPA (Indistinguishable under chosen plaintext attacks) s'il a un avantage négligeable dans le jeu suivant :

1. tout d'abord, le challenger génère un couple de clés (pk, sk) avec pour paramètre de sécurité n, et donne pk à l'attaquant ;
2. l'attaquant peut faire un nombre polynomial d'opérations en le paramètre

de sécurité, dont des chiffrements sous cette clé ;

3. l'attaquant soumet deux messages clairs m_0 et m_1 au challenger ;
4. le challenger choisit au hasard b dans $\{0, 1\}$ et retourne le chiffré $\{m_b\}_{pk}$;
5. l'attaquant peut à nouveau faire un nombre polynomial d'opérations avant de retourner soit 0 ou 1.

L'attaquant gagne le jeu s'il devine correctement la valeur de b. Le chiffrement est IND-CPA si la probabilité qu'un attaquant gagne ce jeu est $1/2 + \epsilon(n)$ avec $\epsilon(n)$ négligeable, c'est-à-dire que pour tout polynôme non nul $p(n)$, il existe n_0 tel que pour tout $n > n_0$, $|\epsilon(n)| < 1/|p(n)|$. Autrement dit, le chiffrement est IND-CPA si l'adversaire n'arrive pas à distinguer de quel message provient un chiffré donné bien qu'il ait choisi les messages à chiffrer.

Exemple de protocole

Présentons un protocole cryptographique, une simplification de la variante à clé publique du protocole de Needham-Schroeder [38] par Roger Needham et Michael Schroeder en 1978 dans le but d'authentifier deux participants entre eux. Les deux participants, que nous appellerons Alice et Bob, possèdent chacun un couple de clés publique/privée, notées (pkA, skA) et (pkB, skB). La notation N utilisée dans le shéma qui suit est ce qu'on appelle des *nonces*, des chaîne de bits générées aléatoirement.

$$A \to B : \{N_A, A\}_{pkB} \tag{1}$$
$$B \to A : \{N_A, N_B\}_{pkA} \tag{2}$$
$$A \to B : \{N_B\}_{pkB} \tag{3}$$

Premièrement, dans (1), Alice envoie à Bob un nonce N_A et son identité A, chiffrés sous la clé publique de Bob pkB. Bob déchiffre ensuite ce message en utilisant sa clé privée skB, et envoie à Alice le message (2) contenant le nonce N_A reçu par Alice pour lui prouver qu'il a bien réussi à ouvrir le message précédent, donc qu'il est bien Bob, ainsi qu'un nouveau nonce N_B, chiffrés sous la clé publique d'Alice pkA. Et finalement, Alice déchiffre ce message en utilisant sa clé privée skA et confirme réception en envoyant à Bob le message (3) contenant le nonce N_B chiffré sous la clé publique de Bob.

Les nonces N_A et N_B sont chiffrés dans tous les échanges sur le réseau, donc ils sont connus uniquement d'Alice et Bob, et Alice et Bob se sont bien assurés qu'ils parlent bien à la bonne personne. Mais est-ce bien vrai ? Supposons qu'Alice effectue le protocole avec Charlie. Charlie est alors capable de se faire passer pour Alice auprès de Bob. C'est un exemple d'attaque d'homme du milieu (MitM : Man in the Middle). L'attaque utilise deux sessions du protocole en

parallèle.

$$A \to C : \{N_A, A\}_{pkC} \tag{1a}$$
$$C \to B : \{N_A, A\}_{pkB} \tag{1b}$$
$$B \to C : \{N_A, N_B\}_{pkA} \tag{2b}$$
$$C \to A : \{N_A, N_B\}_{pkA} \tag{2a}$$
$$A \to C : \{N_B\}_{pkC} \tag{3a}$$
$$C \to B : \{N_B\}_{pkB} \tag{3b}$$

Tout d'abord, dans (1a), Alice envoie un nonce N_A et son identité A à Charlie, sous la clé publique de Charlie. Charlie déchiffre ces informations et les transfère à Bob avec le message (1b). Ce premier message que reçoit Bob est le message d'initiation d'une session du protocole entre Alice et Bob, Bob renvoie donc à Charlie, qu'il croit être Alice, le message (2b). Charlie ne peut pas ouvrir ce message, car il est chiffré sous la clé publique d'Alice. Ce message est transmis par Charlie tel quel à Alice dans (2a). C'est bien le deuxième message auquel Alice s'attendait de la part de Charlie, elle envoie donc le troisième message (3a) contenant N_B chiffré sous la clé de Charlie, qui déchiffre N_B pour le transmettre à Bob dans (3b). Les deux sessions en parallèle finissent correctement, bien que Bob ne parle pas à la bonne personne ! Le protocole ne permet donc pas d'authentifier deux personnes entre elles.

Cette attaque a été publiée pour la première fois par Gavin Lowe [35] en 1996, quasiment vingt ans après son invention. Il obtient cette attaque grâce à des méthodes formelles. Il explique dans cet article une parade pour éviter cette attaque : il suffit de rajouter dans le deuxième message l'identité de Bob, le message devient $\{N_A, N_B, B\}_{pkA}$. Ce message n'est pas déchiffrable par Charlie ; et Alice, en recevant ce message, se rendra compte qu'il ne provient pas de Charlie mais de Bob.

Cet exemple montre qu'il est difficile de savoir, juste en regardant un protocole, s'il est sûr ou pas. Pour être sûr qu'un protocole se comporte bien, il ne suffit pas de le tester et de vérifier que les messages soient bien envoyés. Il faut aussi s'assurer qu'un attaquant n'a pas moyen de contourner le protocole pour obtenir le secret des participants ou de se faire passer pour l'un des participants aux yeux des autres. On ne peut pas savoir cela en testant le protocole, c'est pourquoi, pour chaque protocole, nous devons prouver que les propriétés de sécurité que nous désirons pour ce protocole sont bien correctes.

Contexte

Le domaine de vérification de protocoles cryptographiques est né de ce besoin de prouver la correction de protocoles.

Il y a essentiellement deux modèles pour décrire des protocoles.

– Le modèle de Dolev-Yao [28], aussi appelé modèle symbolique, est un modèle formel simple qui raisonne sur une algèbre de termes. Les messages sont des termes de cette algèbre, et les primitives cryptographiques sont des boîtes noires, des symboles de l'algèbre. L'attaquant peut uniquement

calculer des termes dans l'algèbre, et donc il ne peut utiliser que les primitives cryptographiques définies dans celle-ci. Cela implique que les primitives sont idéales : l'attaquant ne peut pas casser la primitive elle-même, et les nonces sont tous différents. Prouver le secret dans ce modèle se fait en vérifiant que l'attaquant ne peut pas dériver un terme secret.

– Le modèle calculatoire, plus bas niveau, qui raisonne sur les chaînes de bits. Chaque message est une chaîne de bits, et les primitives cryptographiques sont des fonctions polynomiales sur ces chaînes de bits. L'attaquant est une machine de Turing polynomiale probabiliste. Ce modèle est plus réaliste que le modèle formel, parce qu'il prend en compte le fait qu'un attaquant peut casser les primitives, par exemple s'il choisit au hasard une clé pour déchiffrer un message chiffré et qu'il tombe sur la bonne clé, ou bien si il arrive à casser la primitive. Ce modèle permet aussi de quantifier la sécurité d'un protocole par rapport à la sécurité des hypothèses sous-jacentes, la probabilité de collision des nonces, etc. Prouver le secret dans ce modèle est habituellement effectué en montrant qu'un attaquant ne peut pas distinguer le protocole que l'on veut prouver sûr d'un protocole qui est clairement sûr, car le secret n'y apparaît pas. Pour cela, nous utilisons les hypothèses sur les primitives (comme IND-CPA que l'on a vu plus haut).

La sécurité dans le modèle symbolique est plus facile à vérifier, mais une preuve de sécurité dans ce modèle est moins intéressante qu'une preuve dans le modèle calculatoire. Warinschi [49] montre que le protocole Needham-Schroeder-Lowe, la correction du protocole Needham-Schroeder que nous avons présenté ci-dessus, n'est pas sûr dans le modèle calculatoire si le chiffrement utilisé est El Gamal, chiffrement qui est IND-CPA si l'on suppose que le logarithme discret est difficile. Le protocole, pourtant sûr dans le modèle symbolique ne l'est plus dans le modèle calculatoire.

C'est pourquoi de nombreux travaux ont essayé de lier ces deux modèles. Voici un certain nombre de ces travaux :

– le résultat d'Abadi et Rogaway [1] qui montre que, dans le cas du chiffrement symétrique avec quelques hypothèses supplémentaires, lorsqu'on a une preuve dans le modèle formel, on obtient une preuve dans le modèle calculatoire avec un attaquant passif (qui peut uniquement écouter les messages passant sur le réseau, il ne peut pas modifier les messages) ;

– le résultat de Cortier et Warinschi [25] qui montre que la correspondance entre les traces dans les deux modèles dans le cas des protocoles à clé publique, et en présence d'un attaquant actif ;

– le résultat de Comon-Lundh et Cortier [23] qui relie l'équivalence observationnelle dans le pi calcul appliqué (deux processus P et Q sont observationnellement équivalents si, quel que soit le processus O, O en parallèle avec P et O en parallèle avec Q émettent sur les mêmes canaux) aux notions d'indistinguabilité dans le modèle calculatoire ;

– le résultat de Backes [6] qui montre que si une propriété de trace (comme l'authentification) est vraie dans le modèle symbolique, alors, si le protocole n'utilise qu'un certain nombre de primitives cryptographiques (par exemple

du chiffrement à clé publique IND-CCA) et satisfait un certain nombre de conditions, alors cette propriété est vraie aussi dans le modèle calculatoire.

De nombreux outils ont été développés pour obtenir des preuves de sécurité de protocoles. Par exemple, dans le modèle symbolique, il existe actuellement ProVerif [17] développé par Bruno Blanchet, qui prouve automatiquement des protocoles écrits dans le pi calcul appliqué, AVISPA, développé par de nombreux laboratoires, ou Scyther [27], développé par Cas Cremers. Dans le modèle calculatoire, il existe actuellement CryptoVerif [16] par Bruno Blanchet, qui prouve automatiquement des protocoles écrits dans un calcul de processus, un module d'AVISPA pour prouver les protocoles dans le modèle calculatoire [24], ou CertiCrypt et EasyCrypt par Gilles Barthe et al. [7].

Prouver la sécurité d'un protocole est important, mais ce n'est pas suffisant pour s'assurer de la sécurité d'une implémentation d'un protocole. En effet, de nombreux bogues peuvent être présents dans une implémentation, à cause de détails qui ne sont pas traités au niveau de la spécification, ou bien des éléments de la spécification qui n'ont pas été implémentés correctement. C'est pour cela qu'il est important de s'assurer que l'implémentation est aussi correcte.

C'est pour cela que notre but est d'obtenir une implémentation sûre dans le modèle calculatoire d'un protocole cryptographique.

Pour obtenir ceci, il y a essentiellement deux manières de faire :

- on part d'une implémentation d'un protocole, on analyse ce protocole pour obtenir une spécification de celui-ci et enfin prouver que cette spécification est correcte ;

- ou bien on part d'une spécification de protocole, on la prouve correcte et on compile cette spécification en une implémentation.

Nous avons choisi la deuxième méthode pour deux raisons. Principalement, nous pensons qu'il est important qu'avant de commencer toute implémentation d'un protocole, nous soyons sûrs de la sécurité du protocole que nous voulons implémenter, en la prouvant formellement. Si la spécification d'un protocole n'est pas sûre, alors toute implémentation de celui-ci ne sera pas sûre. Par exemple, le protocole TLS (Transport Layer Security) utilisé pour sécuriser des sites web n'a pas moins de six versions différentes (SSL 1.0, SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1, et TLS 1.2) qui ont souvent été conçues pour éviter des failles de sécurité présentes dans les versions précédentes. Cela montre bien la difficulté de concevoir un protocole correct du premier coup, et c'est aussi pourquoi nous aimerions que les concepteurs de protocoles utilisent les outils développés par la communauté de vérification de protocoles pour prouver la correction de leur protocole avant de commencer à l'implémenter. Ensuite, il nous a semblé plus simple de générer des implémentations de protocoles à partir d'une spécification d'un protocole plutôt que d'analyser des implémentations. Analyser des implémentations qui n'ont pas été écrites spécialement pour la vérification est très difficile, peu de méthodes y arrivent.

Figure 1 Travaux en relation

	Symbolique	Calculatoire
Implémentation ↓ spécification	CSur [32] JavaSec [33] ASPIER [22] Dupressoir et al. [29] FS2PV [14] Aizatulin et al. [2] F7, F* [11, 13, 48]	FS2CV [31] F7 [30] Aizatulin et al. [3]
Spécification ↓ implémentation	AGVI [47] χ-spaces [37] Spi2Java [44] JavaSPI [5]	Notre approche [21]

Travaux en relation

Dans la figure 1, nous rassemblons les travaux qui essaient d'obtenir des implémentations de protocole sûres.

De nombreux outils génèrent des implémentations à partir de spécifications.

- L'outil AGVI [47] commence par générer une spécification d'un protocole à partir de propriétés de sécurité qu'on requiert, puis il prouve la correction de ce protocole en utilisant le vérificateur de protocoles Athena, et finalement compile ce protocole en Java.
- L'outil χ-spaces [37] fournit un langage dédié pour spécifier des protocoles, qui peut être ensuite compilé en Java.
- L'outil Spi2Java [46, 44] traduit des protocoles écrit dans le spi-calcul, une extension du pi-calcul pour écrire des protocoles, en Java. La sécurité de la traduction est prouvé dans [44]. Ces protocoles peuvent aussi être vérifiés en utilisant ProVerif ; l'outil a été utilisé sur l'échange de clé du protocole SSH [43].
- Le système JavaSPI [5] est une variante de Spi2Java où le langage de modélisation est Java, et non le spi-calcul.

Tous ces outils diffèrent du notre, parce qu'ils vérifient les protocoles dans le modèle symbolique, alors que nous les vérifions dans le modèle calculatoire.

Les outils suivants analysent les implémentations au lieu de les générer. La plupart d'entre eux n'offrent pas de garantie dans le modèle calculatoire.

- L'outil CSur [32] analyse des protocoles écrits en C en les transformant en clauses de Horn, qui sont ensuite données en entrée au prouveur $\mathcal{H}_1$.
- L'outil JavaSec [33] traduit des programmes en Java en formules logiques du premier ordre, qui sont ensuite données au prouveur de théorèmes du premier ordre e-SETHEO.
- L'outil ASPIER [22] utilise du model checking pour vérifier des implémentations en C de protocoles, en supposant que la taille et le nombre de

sessions est borné. Cet outil a été utilisé pour vérifier la boucle principale d'OpenSSL 3.

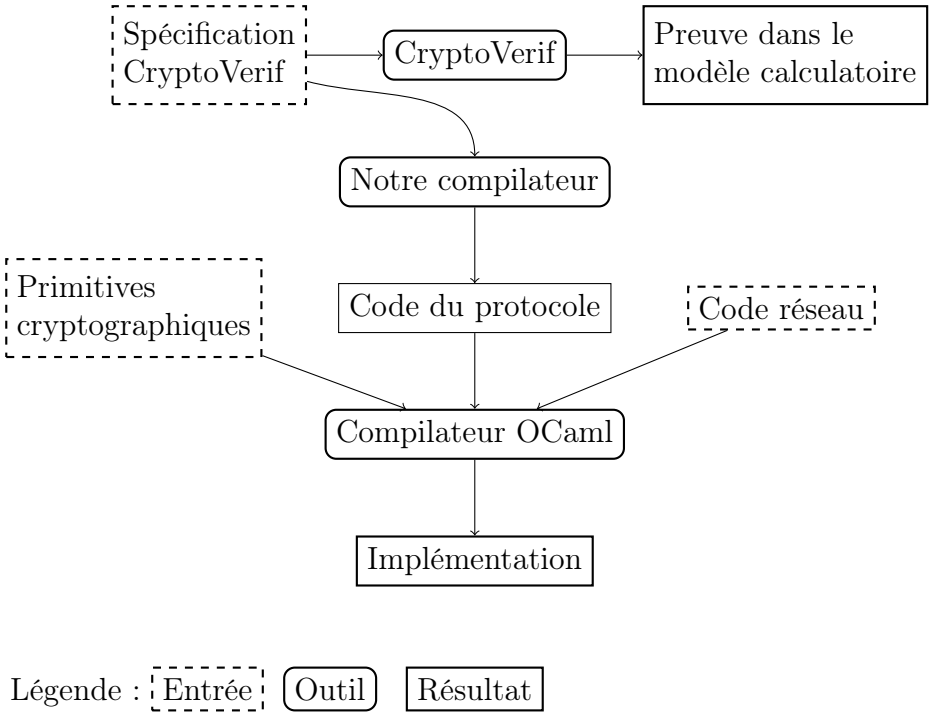
- Dupressoir et al. [29] utilise le vérificateur VCC pour prouver qu'il n'y a pas de mauvais accès mémoire et les propriétés de sécurité de protocoles.
- L'outil FS2PV [14] traduit des protocoles écrits dans un sous-ensemble du langage fonctionnel F# dans le langage d'entrée de ProVerif, qui prouve la correction dans le modèle symbolique. Cette technique est appliquée au protocole TLS [12].
- De manière similaire, Elijah [40] traduit des programmes en Java dans le langage LySa de description de protocoles, qui peut ensuite être vérifié en utilisant LySatool.
- Aizatulin et al. [2] exécutent symboliquement une implémentation d'un protocole en C pour en extraire un modèle ProVerif. Cette technique fonctionne sur des implémentations préexistantes. Par contre, cette technique ne peut analyser qu'un seul chemin dans le protocole, et est donc limité aux protocoles qui n'utilisent pas de branchements. Avec ASPIER [22], c'est l'une des rares méthodes permettant d'analyser des implémentations qui n'ont pas été écrites spécifiquement pour la vérification.
- Les outils F7 et F* [11, 13, 48] utilisent un système de type dépendants pour prouver les propriétés de sécurité de protocoles implémentés en F#, dans le modèle symbolique. L'outil passe à l'échelle, mais il requiert que le programmeur ajoute des annotations de type, qui facilitent la vérification automatique du protocole.

Les outils qui suivent offrent des garanties de sécurité dans le modèle calculatoire.

- L'outil FS2CV [31], similaire à FS2PV, traduit un sous-ensemble de F# dans le langage d'entrée de CryptoVerif, qui peut obtenir une preuve du protocole dans le modèle calculatoire. Cet outil a été appliqué à une petite partie du protocole TLS [12].
- L'outil F7 a aussi été étendu pour obtenir des preuves dans le domaine calculatoire [30], mais il requiert toujours des annotations de type pour aider la preuve.
- L'outil de [2] utilise le résultat de cohérence de [6]. L'idée d'utiliser un résultat de cohérence entre les modèles symboliques et calculatoires peut aussi être utilisé sur d'autres techniques, mais cela réduit la classe des protocoles qui peuvent être utilisés. Pour passer outre ces limitations, les auteurs de [2] ont étendu leur approche pour générer un modèle CryptoVerif [3] pour obtenir des preuves directement dans le modèle calculatoire. La limitation d'un seul chemin d'exécution est toujours présente.

Notre travail complète ces approches en permettant de générer une implémentation d'un protocole au lieu de les analyser.

Figure 2 Vue d'ensemble de notre approche



Vue d'ensemble de notre outil

Nous avons développé un compilateur prenant en entrée une spécification d'un protocole cryptographique écrite dans le langage d'entrée de CryptoVerif et retournant une implémentation en OCaml [39] correspondant à cette spécification.

Nous avons choisi le langage OCaml pour le langage de sortie de notre compilateur pour plusieurs raisons. Premièrement, le langage a une sémantique claire et a un modèle mémoire sûr, ce qui est utile pour prouver la correction du compilateur. OCaml est un langage fonctionnel, ce qui facilite la traduction parce que les oracles définis dans le langage d'entrée de CryptoVerif peuvent être directement traduits dans des fonctions.

La figure 2 présente une vue d'ensemble du fonctionnement de notre approche. Notre approche consiste en plusieurs étapes. Premièrement, nous écrivons dans le langage d'entrée de CryptoVerif la spécification du protocole cryptographique que nous voulons prouver. Cette spécification contient trois éléments :

- la description du protocole dans le langage d'entrée de CryptoVerif ;
- les hypothèses que nous faisons sur les primitives cryptographiques sous-jacentes ;
- les propriétés que nous voulons prouver sur le protocole, comme le secret d'un message ou l'authentification des différents participants.

Il y a deux types d'hypothèses qu'on peut faire sur les primitives sous-jacentes. On peut donner des hypothèses syntaxiques, comme $\mathrm{dec}(\mathrm{enc}(m, k), k) = m$: le déchiffrement d'un chiffré sous la bonne clé donne le clair correspondant, et

on peut donner des hypothèses de sécurité comme le fait que le chiffrement est IND-CPA. Nous utilisons ensuite l'outil CryptoVerif sur cette spécification pour obtenir une preuve dans le modèle calculatoire des propriétés souhaitées.

Ensuite, nous annotons la spécification avec les informations nécessaires pour l'implémentation. Nous utilisons notre compilateur pour obtenir le code du protocole. Tout seul, ce code ne permet pas d'obtenir une implémentation, il nous faut aussi écrire les implémentations des primitives cryptographiques utilisées par le code du protocole. Cette implémentation doit satisfaire les hypothèses que nous faisons sur elles dans la spécification en entrée. Le code du protocole n'est qu'une implémentation des oracles présents dans la spécification, et ne communique pas à travers le réseau. C'est pour cela que nous devons aussi écrire manuellement le code réseau qui s'occupe d'utiliser les traductions des oracles présents dans la spécification et d'envoyer les messages sur le réseau. Le code réseau peut être considéré comme une partie de l'attaquant, et on n'a pas besoin de vérifier sa sécurité. Une fois ces parties écrites, on utilise le compilateur OCaml pour obtenir une implémentation sûre du protocole.

Une implémentation d'un protocole est un ensemble de programmes informatiques qui se parlent entre eux. Si nous prenons l'exemple de Needham-Schroeder présenté ci-dessus, il nous faut trois programmes :

- le programme qui génère les couples de clés publiques et privées pour un participant ;
- le programme d'Alice, qui envoie le message (1), attend la réception du message (2) et enfin envoie le dernier message (3) ;
- le programme de Bob, qui attend la réception du message (1), envoie le message (2) et enfin attend la réception du dernier message (3).

C'est pourquoi nous devons annoter la spécification en la découpant en plusieurs sous-processus, que nous appelons *rôles*. Le compilateur génère ensuite, pour chacun de ces rôles, un module OCaml qui pourra être utilisé par le code réseau.

Nous avons écrit un article de conférence [19], puis un article journal étendant cet article [20], qui présente ce compilateur et une application de celui-ci au protocole SSH Transport Layer Protocol, la première partie du protocole SSH (Secure SHell) qui sert à échanger les clés du tunnel dans lequel les messages seront transmis. Nous y prouvons l'authentification du serveur et le secret des clés échangées.

D'autres travaux ont eu comme but d'obtenir des implémentations vérifiées de SSH. Poll et Schubert [45] ont vérifié une implémentation de SSH en Java en utilisant ESC/Java2, où ESC/Java2 vérifie que l'implémentation ne lance pas d'exceptions et suit la spécification de SSH écrite dans un automate fini, mais ne prouve pas les propriétés de sécurité. L'outil Spi2Java [46, 44] a aussi été utilisé pour obtenir une implémentation prouvée de l'échange de clés du protocole SSH dans le modèle symbolique [43].

Nous avons ensuite écrit l'article [21] qui présente la preuve de correction de notre compilateur, qui n'était pas présente dans [19].

Pour prouver cela, nous avons besoin d'une sémantique formelle d'OCaml. Nous avons adapté la sémantique opérationnelle à petit pas d'une partie du langage par Owens et al. [41]. Nous avons ajouté à ce langage les modules,

multiples fils d'exécution où uniquement un fil peut être exécuté à la fois, et on peut communiquer entre différents fils grâce à de la mémoire partagée.

Un attaquant contre notre implémentation est un programme OCaml qui utilise les modules que nous avons générés. Du côté de CryptoVerif, un attaquant est un processus exécuté en parallèle avec le protocole, qui va essentiellement appeler les oracles à sa disposition et peut faire d'autres calculs. Dans notre preuve, pour chaque attaquant OCaml qui utilise nos modules, on construit un attaquant CryptoVerif contre le protocole de départ qui simule ce que fait l'attaquant OCaml. Lorsque l'attaquant OCaml appelle l'une des fonctions d'un de nos modules générés, l'attaquant CryptoVerif appelle l'oracle correspondant.

Nous établissons ensuite une correspondance précise entre les traces de l'attaquant CryptoVerif en parallèle avec le protocole, d'une part, et les traces de l'attaquant OCaml qui utilise nos modules générés, d'autre part. Cette correspondance nous permet de montrer que la probabilité de succès d'une attaque est la même du côté OCaml et du côté CryptoVerif. C'est pourquoi, si CryptoVerif arrive à prouver la sécurité du protocole, alors, notre implémentation de celui-ci est aussi sûre, et la borne que donne CryptoVerif sur la probabilité de succès d'une attaque peut être adaptée avec les paramètres de l'implémentation pour obtenir une borne sur la sécurité de celle-ci.

Nous avons fait plusieurs hypothèses pour obtenir cette preuve ; les points importants sont :

A1. Les primitives cryptographiques suivent correctement les hypothèses faites sur elles dans la spécification.

A2. Les rôles sont exécutés dans l'ordre spécifié dans la spécification CryptoVerif. Par exemple, dans un protocole d'échange de clés, la génération de clé se fait avant d'appeler les clients et les serveurs.

A3. L'attaquant et le code réseau n'accède pas aux fichiers créés par notre implémentation (comme les fichiers contenant les clés privées).

A4. Le code réseau est un programme OCaml bien typé, qui n'utilise pas de fonctions non sûres pour passer outre le système de typage.

A5. Le code réseau ne modifie pas les chaînes de bits passées à ou reçues de notre code généré. Cette propriété peut être garantie en représentant les chaînes de bits par un type immuable. Mais, le type le plus naturel pour représenter une chaîne de bits est le type `string`, qui est mutable. On peut représenter des chaînes immuables en utilisant un type abstrait. Dans notre sémantique, le type `string` est immuable.

A6. Notre sémantique des fils d'exécutions doit être obéie, ce qui implique que deux programmes qui lisent ou écrivent dans le même fichier ne sont pas exécutés en même temps (on peut faire respecter ceci en utilisant des verrous), et qu'on ne peut pas utiliser *fork* au milieu d'un rôle.

Plan

En partie I, nous introduisons notre compilateur en détail, et expliquons notre application de ce compilateur à la couche de transport du protocole SSH.

Tout d'abord, dans le chapitre 1, nous commençons par décrire l'outil CryptoVerif, puis nous expliquons la syntaxe du langage d'entrée. Nous expliquons ensuite quelles sont les annotations que l'on peut indiquer dans une spécification CryptoVerif afin de pouvoir utiliser notre compilateur. Ces annotations découpent le processus qui décrit le protocole en rôles et indiquent quelle est l'implémentation correspondant à chaque primitive cryptographique. Ensuite, nous parlons des aménagements que nous avons faits à la syntaxe de CryptoVerif. Nous avons introduit le concept de tables, qui permet de modéliser des tables de clés. CryptoVerif utilise un autre concept pour modéliser cela : la construction find. Cette construction est très compliquée à traduire, c'est pour cette raison que nous avons introduit ce nouveau concept qui nous permet de contourner la difficulté. L'outil CryptoVerif, par contre, ne comprend pas cette nouvelle construction, et nous expliquons comment il transforme les tables de clés en find. Nous avons aussi ajouté les macros de fonctions letfun. Nous expliquons enfin nos modifications dans la façon dont CryptoVerif prouve la correction d'un protocole afin d'être capable de prouver l'authentification du serveur et le secret des clés échangées dans la couche de transport de SSH.

Dans le chapitre 2, nous décrivons le langage fonctionnel OCaml. Nous y expliquons sa syntaxe.

Dans le chapitre 3, nous donnons les règles pour traduire un processus CryptoVerif en OCaml. Nous traduisons chaque rôle présent dans la spécification CryptoVerif en entrée dans un module OCaml qui représente ce rôle. Nous expliquons informellement quelles sont les hypothèses que nous devons faire pour que cette traduction soit correcte.

Et finalement, dans le chapitre 4, nous commençons par expliquer comment le protocole SSH fonctionne. Nous entrons dans les détails de la couche de transport, qui contient un échange de clés et la mise en place du tunnel, et nous présentons notre modèle de cette partie du protocole. Nous prouvons ensuite à l'aide de CryptoVerif que ce modèle vérifie bien les propriétés que nous désirons. CryptoVerif est capable de prouver automatiquement l'authentification du serveur, mais nous avons été obligés de donner des indications de preuve pour que CryptoVerif soit capable de prouver le secret des clés échangées. Nous parlons ensuite de l'implémentation que nous générons à partir de ce modèle, et du fait que nous sommes arrivés à la faire interagir avec OpenSSH.

En partie II, nous expliquons la preuve de correction du compilateur.

Nous commençons par introduire dans le chapitre 5 la sémantique du langage d'entrée de CryptoVerif, qui est une relation de réduction sur des configurations. Nous nous limitons au langage sans les annotations pour l'implémentation et sans letfun. Nous ajoutons quelques hypothèses sur la forme de la spécification afin de simplifier la preuve.

Dans le chapitre 6, nous introduisons la sémantique du langage OCaml, qui est le langage de sortie de notre compilateur. Une implémentation d'un protocole est une collection de programmes qui se parlent entre eux, nous avons donc introduit de nouvelles expressions dans le langage afin de pouvoir gérer plusieurs fils d'exécution. Nous avons aussi ajouté une fonction, random, qui nous permet de choisir un booléen au hasard, ce qui rend notre sémantique probabiliste. Nous expliquons ensuite comment nous instrumentons le langage et la sémantique

afin d'effectuer la preuve. Par exemple, ces instrumentations nous permettent de distinguer si une clôture provient d'une de nos fonctions générées ou si elle provient de l'attaquant. Nous y expliquons également comment nous modélisons les fichiers privés générés par notre implémentation.

Nous décrivons les quelques modifications que nous apportons à la traduction dans le chapitre 7 afin d'utiliser l'instrumentation que nous avons introduit dans le chapitre précédent. Nous expliquons aussi comment nous utilisons le modèle de fichiers.

Finalement, le chapitre 8 explique la preuve en elle-même. Nous commençons par introduire la formalisation de l'hypothèse A1 et nous prouvons que, si une primitive se comporte comme il faut dans un environnement vide, alors elle se comporte aussi comme il faut lorsqu'elle est appelée dans n'importe quel environnement. Pour cela, l'hypothèse A5 est importante : ce qui est donné à la primitive ne peut pas être modifié ensuite. Ensuite, nous prouvons que la traduction est correcte : le code OCaml généré à partir d'un processus CryptoVerif se comporte de la même manière que celui-ci. Plus précisément, nous prouvons que lorsqu'une configuration CryptoVerif se réduit en une autre configuration, les configurations OCaml qui correspondent se réduisent de la même manière. Nous complétons cette preuve en annexe A. Nous introduisons ensuite le simulateur, un attaquant CryptoVerif qui simule l'attaquant OCaml. C'est une boucle qui commence par appeler une fonction de simulation, puis qui réagit en fonction du résultat de cette fonction. La fonction de simulation prend en entrée la configuration OCaml à simuler, et évalue cet état avec la sémantique d'OCaml jusqu'à qu'on appelle une de nos fonctions générées. La fonction retourne alors la configuration dans laquelle elle s'est arrêtée, ainsi que l'oracle correspondant à la fonction. Le simulateur s'occupe alors d'appeler cet oracle, puis de calculer une nouvelle configuration OCaml à donner au simulateur en fonction du résultat de l'oracle. Il continue en entrant à nouveau dans la boucle avec ce nouvel état. Une fois ce simulateur introduit, nous pouvons commencer à relier les deux attaquants. Nous introduisons la sémantique intermédiaire pour le simulateur, qui est la sémantique de CryptoVerif où l'on précise aussi les réductions OCaml à l'intérieur de la fonction de simulation. Nous introduisons ensuite une relation entre cette sémantique intermédiaire et la sémantique d'OCaml. Nous prouvons en annexe B le fait que cette relation est bien gardée après réduction. Enfin, nous expliquons ce que ce résultat implique quant à la sécurité de l'implémentation.

Notations

Introduisons quelques notations utilisées dans la suite. Lorsque f est une fonction, on note $\mathrm{Dom}(f)$ le domaine de f, c'est-à-dire l'ensemble des éléments x tel que $f(x)$ est défini. On note $f[x \mapsto y]$ la fonction f' définie par $f'(x) = y$ et $f'(x') = f(x')$ pour $x \neq x'$. Lorsque les fonctions f_1 et f_2 ont leurs domaines disjoints, on note $f_1 \cup f_2$ la fonction f' définie par $f'(x) = f_1(x)$ lorsque $x \in \mathrm{Dom}(f_1)$ et $f'(x) = f_2(x)$ lorsque $x \in \mathrm{Dom}(f_2)$. Lorsque f_1 et f_2 sont des fonctions, la notation $f_1 \subseteq f_2$ (ou $f_2 \supseteq f_1$) signifie que $\mathrm{Dom}(f_1) \subseteq \mathrm{Dom}(f_2)$ et pour tout $x \in \mathrm{Dom}(f_1)$, on a $f_2(x) = f_1(x)$. On note $\emptyset$ la fonction qui a pour

domaine l'ensemble vide $\emptyset$.

Introduction

Cryptography is presently used in many contexts in order to send messages in a secure way. When we pay a merchant with our credit card, encrypted data is transmitted between the card, the merchant, and the bank. These transmissions ensure that the card is not fraudulent, and allow the bank to carry out the requested transaction. The way messages are transmitted between all the parties is what we call a cryptographic protocol.

The security of a protocol depends on multiple properties, like

- confidentiality: no third party observing the communication can deduce the contents of the messages,
- authentication: when a party receives a message, he can be sure that the message has been sent by its alleged sender, and not by someone posing as him,
- integrity: we are sure a message we received is the same as the message sent by our interlocutor.

Cryptographic Primitives

A protocol uses cryptographic primitives in order to ensure its security. In order to understand what cryptography is capable of, let us present some primitives used in these protocols.

In order to ensure confidentiality, most protocols use encryption. Symmetric encryption is a way of transforming a message so that it is unreadable for anybody that does not possess the key used to encrypt the message. Symmetric encryption is composed of three algorithms: the key generation algorithm, which takes a security parameter (e.g., the size of the key to generate) and generates a new key k; the encryption algorithm, which takes a message m and a key k, and returns the ciphertext $\{m\}_k$; and the decryption algorithm, which takes a ciphertext $\{m\}_k$ and the key k used to encrypt the ciphertext, and returns the plaintext m. In order to use symmetric encryption, both parties must know a common secret, the encryption key, before being able to exchange encrypted messages.

Asymmetric encryption, or public key encryption, allows one to encrypt a message for a given party without knowing a common secret with him. Similarly to symmetric encryption, asymmetric encryption is composed of three algorithms. The key generation algorithm generates a new pair of keys (pk, sk). The key pk

is the public key and sk is the private key. The public key is only used to encrypt and the secret key is only used to decrypt. The encryption algorithm takes a message m and a public key pk, and returns a ciphertext $\{m\}_{pk}$. This ciphertext depends on random bits: a message encrypted multiple times under the same key gives different ciphertexts. The decryption algorithm takes a ciphertext $\{m\}_{pk}$ and the private key sk corresponding to the public key pk, and returns the plaintext m. The public key is meant to be published to all our potential interlocutors, but the private key must be kept secret: if a third party has access to this key, he will be able to decrypt all the messages sent to us.

Signatures are used to ensure authentication and integrity. Signature is also composed of three algorithms. The key generation algorithm generates a pair of keys (pk, sk), sk is the signature key and pk is the verification key, used to check the signatures. The signature algorithm takes a message m and a signature key sk, and returns the signature $\{m\}_{sk}$ corresponding to message m. The signature verification algorithm takes a message m, a signature s, and a verification key pk. If the signature s is a correct signature of message m under the signature key sk corresponding to pk, then the algorithm returns that the signature is correct, and otherwise returns that the signature is incorrect. Similarly to asymmetric encryption, the signature key sk is meant to be kept secret, and the key pk must be published to every party that would want to verify our signature.

Hash functions are functions that take a message m and return a bitstring of fixed length. This bitstring is called the hash of the message m. Hash functions are one-way functions: one can easily obtain the hash of a message, we just apply the function; but it is hard, given a hash h, to find a message m such that its hash is h.

Message Authentication Codes (MAC) are functions that take a message m and a key k, and return a hash corresponding to these two elements. The goal of this primitive is to ensure integrity of messages: the key k allows, in a manner, to choose the hash function to apply to a message, and this hash function is unknown for anyone not knowing the key. A third party cannot create the MAC corresponding to a message m without also knowing the key k.

The security of these primitives is defined by games played between a challenger and an adversary, which is a polynomial probabilistic Turing machine. The game depends on the primitive and on the security property we want. For example, asymmetric encryption is IND-CPA (Indistinguishable under chosen plaintext attacks) when the adversary has a negligible advantage in the following game:

1. the challenger generates a pair of keys (pk, sk) with security parameter n, and gives pk to the adversary,

2. the adversary can do a polynomial number of operations (encryptions under this key is also an operation) in the security parameter,

3. the adversary submits to the challenger two plaintexts m_0 and m_1,

4. the challenger chooses randomly b in $\{0, 1\}$ and returns the ciphertext $\{m_b\}_{pk}$,

5. the adversary can again do a polynomial number of operations before returning 0 or 1.

The adversary wins the game when he correctly guesses the value of b. The encryption is IND-CPA when the probability that an adversary wins this game is $1/2 + \epsilon(n)$ where $\epsilon(n)$ is negligible, that is, for all non-null polynomial $p(n)$, there exists n_0 such that for all $n > n_0$, $|\epsilon(n)| < 1/|p(n)|$. Put another way, encryption is IND-CPA when the adversary is not able to distinguish from which plaintext a ciphertext is coming from, even when he chooses the plaintexts to encrypt.

Protocol Example

Let us present a simplification of the public key variant of the Needham-Schroeder protocol [38] by Roger Needham and Michael Shroeder in 1978, whose goal is to authenticate two parties. Both participants, that we will call Alice and Bob, possess a pair of public/private keys, that we denote (pkA, skA) and (pkB, skB). The notation N in the following scheme denotes *nonces*, randomly generated bitstrings.

$$A \to B : \{N_A, A\}_{pkB} \tag{1}$$
$$B \to A : \{N_A, N_B\}_{pkA} \tag{2}$$
$$A \to B : \{N_B\}_{pkB} \tag{3}$$

First, in (1), Alice sends to Bob a nonce N_A and its identity A, encrypted under the public key of Bob pkB. Bob then decrypts this message using his private key skB, and sends back to Alice the message (2) containing the nonce N_A received from Alice in order to prove to her he was able to open the previous message, and so that he is, in fact, Bob, and also a new nonce N_B, both encrypted under the public key of Alice pkA. Finally, Alice decrypts this message using her private key skA and confirms reception to Bob by sending the message (3) containing the nonce N_B encrypted under the public key of Bob.

The nonces N_A and N_B are encrypted in all transmissions over the network, and so they are only known by Alice and Bob, and they make sure they were talking to the correct participant. But is it really true? Let us assume that Alice performs the protocol with Charlie. Charlie is then able to pose as Alice in the eyes of Bob. This is an example of Man in the Middle attack (MitM). The attack uses two sessions of the protocol in parallel.

$$A \to C : \{N_A, A\}_{pkC} \tag{1a}$$
$$C \to B : \{N_A, A\}_{pkB} \tag{1b}$$
$$B \to C : \{N_A, N_B\}_{pkA} \tag{2b}$$
$$C \to A : \{N_A, N_B\}_{pkA} \tag{2a}$$
$$A \to C : \{N_B\}_{pkC} \tag{3a}$$
$$C \to B : \{N_B\}_{pkB} \tag{3b}$$

First, in (1a), Alice sends a nonce N_A and her identity A to Charlie, under the public key of Charlie. Charlie decrypts this information and sends it over to Bob in message (1b). This first message received by Bob is the initial message of a protocol session between Alice and Bob, and so Bob sends back the message (2b) to Charlie, who Bob thinks to be Alice. Charlie cannot open this message because it is encrypted under the public key of Alice. Charlie sends this message verbatim to Alice in (2a). This is indeed the second message Alice expected from Charlie, she sends back the third message (3a) containing N_B encrypted under the public key of Charlie. Charlie then decrypts N_B and sends it to Bob in (3b). Both sessions finish correctly, but Bob does not talk to the intended party! The protocol is not able to authenticate our two parties.

This attack has been published for the first time by Gavin Lowe [35] in 1996, almost twenty years after the invention of the protocol. He obtained this attack with the help of formal methods. He explains in this article a way to avoid this attack: we add to the second message the identity of Bob, the message then is $\{N_A, N_B, B\}_{pkA}$. This message is not decryptable by Charlie, so Alice will notice that the message did not come from Charlie but from Bob.

This example shows that it is difficult to know, just by looking at a protocol, whether it is secure or not. To be sure that a protocol is secure, testing that messages are correctly sent and received is not sufficient. We must also ensure that an adversary has no means to bypass the protocol in order to obtain secret keys or to pose as another party in the eyes of the other parties. That is why we must prove, for each protocol, that the security properties we want are indeed correct.

Context

The domain of cryptographic protocol verification is born from this need to prove the correctness of protocols.

There are essentially two models to describe protocols.

- The Dolev-Yao model [28], also known as the symbolic model, is a simple formal model that works on a term algebra. Messages are terms of this algebra, and cryptographic primitives are black boxes, symbols of the algebra. The adversary can only calculate terms in the algebra, so he can only use the primitives defined in it. Primitives are ideal: the adversary cannot break the primitive itself, and nonces are all different. Proving the secrecy in this model is proving that the adversary cannot deduce a secret term.

- The computational model, more low-level, works with bitstrings. A message is a bitstring, cryptographic primitives are polynomial functions on these bitstrings. The adversary is a polynomial probabilistic Turing machine. This model is more realistic than the formal model, because it takes into account the fact that an adversary can break a primitive, or guess the encryption key for example. This model is able to quantify the security of a protocol in relation to the security of underlying assumptions, the

probability of collision of nonces, etc. Proving the secrecy in this model is usually done by proving that the protocol that we want to prove secure is indistinguishable from a protocol that is clearly secure, because the secret is not present in it. These proofs use assumptions on primitives (like IND-CPA) in order to conclude.

Security in the symbolic model is simpler to verify, but a proof of security in this model is less interesting than a proof in the computational model. Warinschi [49] shows that the Needham-Schroeder-Lowe protocol, the correction of the Needham-Schroeder protocol we have presented above, is not secure in the computational model if the encryption we use is El Gamal, which is a IND-CPA encryption when we assume that discrete logarithm is difficult. The protocol, which is secure in the symbolic model, is not secure anymore in the computational model.

That is why numerous works try to link these models. Let us present some of them:

- the result of Abadi and Rogaway [1] shows that, in the case of symmetric encryption with some more assumptions, when we have a proof in the symbolic model, we have a proof in the computational model with a passive adversary (he can only listen to the network and not alter messages),
- the result of Cortier and Warinschi [25] shows the correspondence between traces in both models for asymmetric encryption with an active attacker,
- the result of Comon-Lundh and Cortier [23] links observational equivalence in the applied pi calculus (two processes P and Q are observationally equivalent when, for every process O, O in parallel with P and O in parallel with Q output on the same channels) to indistinguishability notions in the computational model,
- the result of Backes [6] shows that if a trace property (e.g., authentication) holds in the symbolic model, then it also holds in the computational model, provided the protocol uses only cryptographic primitives in a certain set (e.g., IND-CCA public-key encryption) and satisfies certain soundness conditions.

Numerous tools have been developed to obtain proofs of security. For example, in the symbolic model, there are ProVerif [17], written by Bruno Blanchet, which proves automatically protocols written in applied pi calculus, AVISPA, developed by numerous laboratories, or Scyther [27], written by Cas Cremers. In the computational model, there are CryptoVerif [16] by Bruno Blanchet, which proves automatically protocols written in a process calculus, a module of AVISPA to prove protocols in the computational model [24], or CertiCrypt and EasyCrypt by Gille Barthe et al. [7].

However, proving specifications of protocols in such models is not sufficient. Even if the specification is correct, an implementation of the protocol may be insecure, because of errors in implementation details left unspecified at the specification level, or because the specification has not been correctly implemented.

Figure 3 Related work

	Symbolic	Computational
Implementation ↓ Specification	CSur [32] JavaSec [33] ASPIER [22] Dupressoir et al. [29] FS2PV [14] Aizatulin et al. [2] F7, F⋆ [11, 13, 48]	FS2CV [31] F7 [30] Aizatulin et al. [3]
Specification ↓ Implementation	AGVI [47] χ-spaces [37] Spi2Java [44] JavaSPI [5]	Our work [21]

It is therefore important to make sure that the implementation is secure, and not only the specification.

Hence our goal is to obtain protocol implementations secure in the computational model.

To reach this goal, there are essentially two ways to proceed:

- we start with an implementation of a protocol, we analyze this protocol to obtain a specification of it, and finally we prove that this specification is correct,
- or we start with a specification of a protocol, we prove this specification correct, and then we compile this specification into an implementation.

We chose the latter way for two reasons. First, we believe that starting by designing a protocol, formalizing it, proving it secure formally, and only after that implementing it, is a better methodology than starting from the implementation. If the specification is not secure, then any implementation based on it will be insecure. For example, the TLS (Transport Layer Security) protocol used to secure web sites has no less than six different versions (SSL 1.0, SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2) which have, most of the time, been designed to avoid a security flaw in the protocol in previous versions. This shows how hard it is to design a correct protocol, and this is also why we would like protocol designers to use tools developed by the protocol verification community to verify the correctness of their protocols before implementing them. Second, generating protocol implementations is also somewhat easier than analyzing them; analyzing existing protocol implementations not written for verification is especially difficult, and very few methods can do that.

Related Work

In Figure 3, we put together works that try to obtain secure implementations of cryptographic protocols.

Several tools already use the approach of generating an implementation from a specification.

- AGVI [47] first generates a protocol from security requirements, proves its correctness using the protocol verifier Athena, then compiles the protocol into Java.

- χ-spaces [37] provide a domain-specific language for specifying protocols, which can be interpreted or compiled to Java.

- Spi2Java [46, 44] translates spi-calculus protocols into Java implementations; the soundness of this translation is proved in [44]. The protocols can also be verified using the automatic protocol verifier ProVerif. Spi2Java has been applied to the key exchange part of the SSH Transport Layer Protocol [43].

- The JavaSPI framework [5] is a variant of Spi2Java in which the modeling language is also Java itself, instead of the spi calculus.

All these approaches differ from our work in that they verify protocols in the symbolic model, while we verify them in the more realistic computational model.

The following tools analyze implementations instead of generating them. Many of these approaches do not provide computational security guarantees.

- The tool CSur [32] analyzes protocols written in C by translating them into Horn clauses, given as input to the $\mathcal{H}_1$ prover.

- Similarly, JavaSec [33] translates Java programs into first-order logic formulas, given as input to the first-order theorem prover e-SETHEO.

- ASPIER [22] uses software model-checking to verify C implementations of protocols, assuming the size of messages and the number of sessions are bounded. This tool has been used to verify the main loop of OpenSSL 3.

- Dupressoir et al. [29] use the general-purpose C verifier VCC to prove both memory safety and security properties of protocols.

- The tool FS2PV [14] translates protocols written in a subset of the functional language F# into the input language of ProVerif, to prove them in the symbolic model. This technique was applied to the protocol TLS [12].

- Similarly, Elijah [40] translates Java programs into LySa protocol specifications, which can be verified by the LySatool.

- Aizatulin et al. [2] use symbolic execution in order to extract ProVerif models from pre-existing protocol implementations in C. This technique currently analyzes a single execution path of the protocol, so it is limited to protocols without branching. Together with ASPIER [22], it is one of the rare methods that can analyze implementations not written specifically for verification.

- The tools F7 and F* [11, 13, 48] use a dependent type system in order to prove security properties of protocols implemented in F#, in the symbolic model. This approach scales well to large implementations but requires type annotations, which facilitate automatic verification.

In contrast, the following approaches provide computational security guarantees.

- Similarly to FS2PV, the tool FS2CV [31] translates a subset of F# to the input language of CryptoVerif, which can then provide a proof of the protocol in the computational model. This tool has been applied to a very small subset of the TLS protocol [12].

- The F7 approach has also been extended to the computational model [30], but still requires type annotations to help the proof.

- The tool of [2] provides computational security guarantees by applying the computational soundness result of [6]. However, this restricts the class of protocols that can be considered. To overcome this limitation, the authors of [2] have recently extended their approach to generate a CryptoVerif model [3], thus getting proofs directly in the computational model, still with the limitation to a single execution path.

Our work nicely complements these approaches by allowing one to generate implementations instead of analyzing them.

Overview of our tool

We have developed a compiler taking a specification of a cryptographic protocol written in the input language of CryptoVerif and returning an implementation of it in OCaml [39].

We chose the OCaml language for several reasons, starting with the fact that it is memory safe and has a clean semantics, which is useful to prove the correctness of the compiler. OCaml is a functional language, which facilitates the compilation because the CryptoVerif specification uses oracles that can be immediately translated into functions.

Figure 4 presents an overview of our approach. Our approach consists of multiple steps. First, we write the specification of the protocol we want to prove in the input language of CryptoVerif. This specification contains three elements:

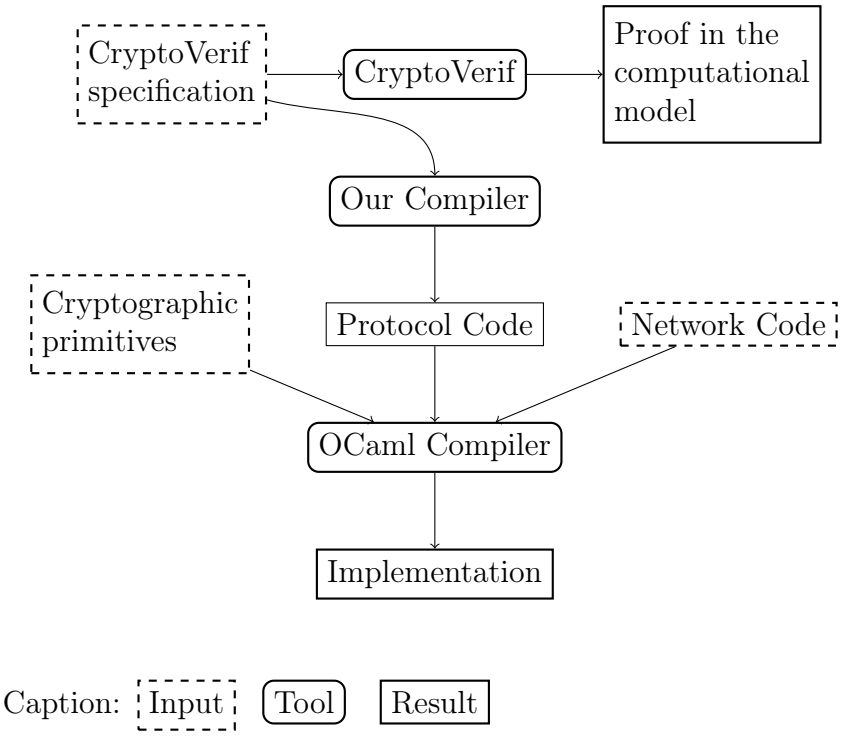
- the description of the protocol in the input language of CryptoVerif,

- the assumptions on the underlying cryptographic primitives,

- the properties we want to prove on the protocol, for example the secrecy of a message or authentication.

There are two types of assumptions one can give on the underlying primitives. One can give syntactic assumptions, like $\mathrm{dec}(\mathrm{enc}(m, k), k) = m$: decryption

Figure 4 Overview of our tool



of the ciphertext under the correct key yields the plaintext, and one can give security assumptions, like encryption is IND-CPA. We use then CryptoVerif on this specification in order to obtain a proof of correctness of the desired properties in the computational model.

Then, we annotate the specification with the required information in order to obtain an implementation. We use our compiler to obtain the protocol code. Alone, this code cannot yield an implementation, we also need to write the implementations of the cryptographic primitives used by the protocol code. These implementations must satisfy the assumptions we have done on them in the specification. The protocol code is only the implementation of the oracles, and it does not communicate through the network. That is why we must also write the network code, which will call the translations of the oracles in the specification and send messages on the network. The network code can be considered as part of the adversary, we do not need to verify that it is secure. As soon as these parts are written, we can use the OCaml compiler to obtain a secure implementation of a protocol.

The implementation of a protocol is a collection of programs talking to each other. Let us illustrate this with the Needham-Schroeder protocol presented above. We need three programs:

- the program that generates a pair of public and private keys for a party,
- the program of Alice, which sends the message (1), waits until reception of message (2) and finally sends back the last message (3),
- the program of Bob, which waits until reception of message (1), sends

message (2) and finally waits until reception of the last message (3).

That is why we must annotate the specification in order to separate it in multiple sub-processes that we name *roles*. Our compiler will then generate, for each role present in the specification, an OCaml module that can in turn be used by the network code.

We have written a conference article [19], and a journal article [20] presenting this compiler and its application to the SSH Transport Layer Protocol, the first part of the SSH (Secure SHell) protocol that exchanges the keys of the tunnel where messages will be sent between parties. We prove the authentication of the server and the secrecy of exchanged keys.

Other works obtain verified implementations of SSH. Poll and Schubert [45] verified an implementation of SSH in Java using ESC/Java2: ESC/Java2 verifies that the implementation does not raise exceptions, and follows a specification of SSH by a finite automaton, but does not prove security properties. Spi2Java [46, 44] has also been in order to obtain a proved implementation of the key exchange of SSH in the symbolic model [43].

We then wrote the article [21], which presents the proof of correctness of our compiler, which was not present in [19].

To make this proof, we needed a formal semantics of OCaml. We adapted the operational small-step semantics of a core part of OCaml by Owens et al. [41]. We added to this language support for simplified modules, multiple threads where only one thread can run at any given time, and communication between threads by a shared part of the store.

An adversary against the generated implementation is an OCaml program using the modules generated by our compiler. On the CryptoVerif side, an adversary is a process running in parallel with the verified protocol, which will essentially call the available oracles and can do other calculations. In our proof, for each OCaml adversary, we construct a corresponding CryptoVerif adversary that simulates the behavior of the OCaml adversary. When the OCaml adversary calls one of the functions generated by our compiler, which comes from an oracle in the CryptoVerif process, the CryptoVerif adversary calls this oracle. Then we establish a precise correspondence between the traces of the CryptoVerif process in parallel with that CryptoVerif adversary, on the one hand, and the traces of the OCaml program using our generated modules, on the other hand. This correspondence allows us to show that the probability of success of an attack is the same on the CryptoVerif side and on the OCaml side. Therefore, if CryptoVerif proves that the protocol is secure, then the generated OCaml implementation is also secure, and the bound on the probability of success of an attack computed by CryptoVerif is also valid for the implementation.

We have made several assumptions to obtain this proof; the most important ones are:

A1. The cryptographic primitives are correct with respect to the assumptions made on them in the specification.

A2. The roles are executed in the order specified in CryptoVerif (e.g., in a key-exchange protocol, the key generation is called before the servers and clients).

A3. The adversary and the network code do not access files created by our implementation (e.g., private key files).

A4. The network code is a well-typed OCaml program, which does not use unsafe OCaml functions to bypass the type system.

A5. The network code does not mutate bitstrings passed to or received from generated code. This property can be guaranteed by representing bitstrings by an immutable OCaml type. However, the most natural type for representing bitstrings is the OCaml type `string`, which is mutable. Immutable strings can be implemented in OCaml using an abstract type instead of `string`. In our semantics, strings are immutable values.

A6. Our semantics of threads is obeyed, which implies that two processes that read or write the same file are not run concurrently (which can be enforced using locks), and that one cannot fork in the middle of a role.

Plan

In Part I, we introduce our compiler in detail, and we explain its application to the SSH Transport Layer Protocol.

First, in Chapter 1, we describe the CryptoVerif tool and we explain the syntax of the input language. Then, we explain the annotations one can put on a CryptoVerif specification in order to use our compiler. These annotations separate the process that describes the protocol in roles and link cryptographic primitives to their implementation. Then we present the improvements we did on the CryptoVerif syntax. We introduce the concept of tables, in order to model key tables. CryptoVerif uses another concept to model this: the find construct. This construct is really hard to translate, that is why we have introduced this new concept in order to bypass the difficulty. The CryptoVerif tool, internally, does not understand this new construct, and we explain how it transforms tables to find. We have also added function macros letfun. Finally, we explain our improvements to the way CryptoVerif proves protocols in order to be able to prove the authentication and the secrecy of exchanged keys in the transport layer of SSH.

In Chapter 2, we describe the functional language OCaml. We explain its syntax.

In Chapter 3, we give the translation rules in order to translate a CryptoVerif process to OCaml. We translate each role present in the CryptoVerif specification to an OCaml module corresponding to this role. We explain informally the assumptions we must have in order to have a correct implementation.

Finally, in Chapter 4, we explain how the SSH protocol works. We detail the transport layer, which contains a key exchange and sets up a tunnel, and we present our model of this part of the protocol. We then prove with the help of CryptoVerif that this model is secure. CryptoVerif is able to automatically prove the authentication of the server, but we needed to give it proof indications in order to prove the secrecy of the exchanged keys. Then we talk about the

implementation we generated from this model, and mention that we were able to make our implementation interact with OpenSSH.

In Part II, we explain the proof of correctness of the compiler.

In Chapter 5, we begin by introducing the semantics of the input language of CryptoVerif, which is a reduction relation on configurations. We limit ourselves to the language without annotations for the implementation, and without letfun. We add some more assumptions on the form of the specification in order to simplify the proof.

In Chapter 6, we introduce the semantics of the OCaml language, which is the output language of our compiler. An implementation of a protocol is a collection of programs that talk to each other, so we introduced new expressions in the language to manage multiple execution threads. We also added the function random, which chooses a boolean randomly. Our semantics become probabilistic. We explain how we instrument the language and the semantics in order to obtain the proof. For example, these instrumentations allow us to distinguish whether a closure comes from one of our generated functions or the adversary. We explain how we model private files generated by our implementation.

We describe the modifications to the translation in Chapter 7 in order to take into account the instrumentation we described in the previous chapter. We also explain how we model files.

Finally, Chapter 8 explains the proof of correctness. We begin by formalizing Assumption A1 and we prove that, when a primitive behaves correctly in an empty environment, it also behaves correctly when called in any environment. To prove this, Assumption A5 is important: what is given to the primitive cannot be modified afterwards. Then, we prove that the translation is correct: the OCaml code generated from a given CryptoVerif process behaves the same as this process. More precisely, we prove that when a CryptoVerif configuration reduces in another configuration, the corresponding OCaml configurations reduce in the same manner. We complete this proof in Appendix A. We introduce the simulator, a CryptoVerif adversary that simulates the OCaml adversary. It is a loop that begins by calling the simulation function, and then reacts according to the result of this function. The simulation function takes as argument the OCaml configuration to simulate, and evaluates it using the OCaml semantics until it encounters one of our generated functions. The function then returns the current configuration and the oracle to call. The simulator then calls the oracle corresponding to the function, and transforms the OCaml configuration by integrating into it the result returned by the oracle, and reenters the loop. We can now link these two adversaries. We introduce the intermediate semantics for the simulator, which is the CryptoVerif semantics where we precise the OCaml reductions in the simulation function. We introduce then a relation between the intermediate and OCaml semantics. We prove in Appendix B that this relation is kept after reduction. We finally explain what this result implies for the security of the implementation.

Notations

Let us introduce some basic notations. When f is a function, we denote by $\mathrm{Dom}(f)$ the domain of f, that is, the set of elements x such that $f(x)$ is defined. We denote by $f[x \mapsto y]$ the function f' defined by $f'(x) = y$ and $f'(x') = f(x')$ for $x' \neq x$. When f_1 and f_2 are functions with disjoint domains, we denote by $f_1 \cup f_2$ the function f' defined by $f'(x) = f_1(x)$ if $x \in \mathrm{Dom}(f_1)$ and $f'(x) = f_2(x)$ if $x \in \mathrm{Dom}(f_2)$. When f_1 and f_2 are functions, we write $f_1 \subseteq f_2$ (or $f_2 \supseteq f_1$) when $\mathrm{Dom}(f_1) \subseteq \mathrm{Dom}(f_2)$ and, for all $x \in \mathrm{Dom}(f_1)$, we have $f_2(x) = f_1(x)$. We denote by $\emptyset$ the function whose domain is the empty set $\emptyset$.

Part I

Compiler Description and Applications

Chapter 1

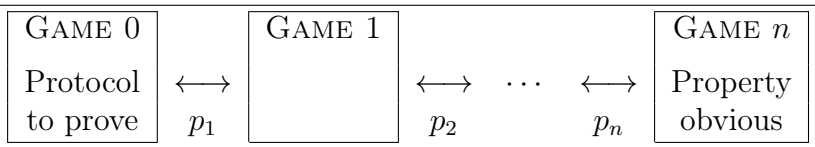
CryptoVerif

1.1 Description of the Tool

The CryptoVerif tool verifies cryptographic protocols in the computational model. To prove the correctness of a protocol, the tool needs:

- the representation of the protocol, written in a process calculus,
- the functional assumptions (e.g., the decryption of the ciphertext is the plaintext) and the security assumptions (e.g., the encryption is IND-CPA),
- and the properties we want to prove, for example the secrecy of a variable or authentication properties.

Figure 1.1 Sequence of games



Probabilities $p_1, \ldots, p_n$ are negligible.

To prove that the properties are correct for the given protocol, the tool generates a sequence of games, like the manual proofs written by cryptographers. Figure 1.1 is a graphical representation of the sequence of games. The first game is the process of the protocol we want to prove, the last game is a game where the properties we want to prove are obvious, and two consecutive games are distinguishable only with negligible probability. So the properties are also true in the first game with overwhelming probability. The games are written in a probabilistic polynomial-time process calculus.

The tool defines transformations to apply to games to obtain the next game. Arguably, the most important one would be the transformation that applies a cryptographic assumption. For example, suppose we have added the assumption that the symmetric encryption is IND-CPA, that is to say, any polynomial adversary cannot distinguish between a ciphertext of a message m encrypted under a key k he does not know and the encryption under the key k of the message

of same length as m containing only zeroes. The cryptographic transformation corresponding to IND-CPA would transform an encryption of a message m under a key k not known to the adversary to an encryption of the message of the same length containing only zeroes.

By default, the tool automatically finds which transformation would be the most interesting to apply on the current game and applies it, until it reaches a game that satisfies the properties we want to prove. Sometimes, this strategy does not work. In these cases, one can also manually indicate which transformations to apply.

The process calculus used by games is subtly different from the process calculus used to describe a protocol. There are presently two such languages.

Channel front-end. This language is closer to the internal language of CryptoVerif. It is a probabilistic process calculus using channels to pass information between processes.

Oracle front-end. This language represents more closely what a cryptographer would write. One declares oracles that can be called by the adversary. Instead of waiting on a channel, we declare an oracle, and instead of writing on a channel, we return the result of the oracle.

The oracle front-end is more adapted to be translated in a functional programming language, because oracles are similar to functions. We present this language in the next section.

1.2 Protocol Representation Language

The protocol is represented in the language of Figure 1.2.

This language uses types denoted by T, which are subsets of $\mathit{bitstring}_\perp = \mathit{bitstring} \cup \{\perp\}$ where *bitstring* is the set of all bitstrings and $\perp$ is a special symbol (used for example to represent the failure of a decryption). Some types are predefined: $\mathit{bool} = \{\mathrm{true}, \mathrm{false}\}$, where false is 0 and true is 1; *bitstring*; and $\mathit{bitstring}_\perp$.

A bitstring b represents a concrete value. It cannot be used in the protocol, but is useful when describing the adversary.

Each variable x is an array, so that CryptoVerif can refer to any value a variable has taken. The notation $\widetilde{i}$ denotes a tuple $i_1, \ldots, i_m$ of array indices.

The language also uses function symbols f. Each function symbol comes with a type declaration $f : T_1 \times \ldots \times T_m \to T$, and represents an efficiently computable, deterministic function that maps each tuple in $T_1 \times \ldots \times T_m$ to an element of T. Particular functions are predefined, and some of them use the infix notation: $M = N$ for the equality test, $M \neq N$ for the inequality test, $M \vee N$ for the boolean or, $M \wedge N$ for the boolean and, $\neg M$ for the boolean negation.

In this language, terms represent computations on bitstrings. The term $x[\widetilde{i}]$ evaluates to the content of the variable x with indices $\widetilde{i}$. We use x, y, z, u as variable names. The function application $f(M_1, \ldots, M_m)$ returns the result of applying the function f to $M_1, \ldots, M_m$.

Figure 1.2 Protocol representation language

$M, N ::=$	terms
a	bitstring
$x[i_1, \ldots, i_m]$	variable
$f(M_1, \ldots, M_m)$	function application
$Q ::=$	oracle declarations
0	nil
$Q \mid Q'$	parallel composition
foreach $i \leq n$ do Q	replication n times
$O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P$	oracle declaration
$P ::=$	oracle body
return$(M_1, \ldots, M_k); Q$	return
end	end
$x[\widetilde{i}] \stackrel{R}{\leftarrow} T; P$	random number
$x[\widetilde{i}] \leftarrow M; P$	assignment
if M then P else P'	conditional
event $e(M_1, \ldots, M_k); P$	event
insert $Tbl(M_1, \ldots, M_k); P$	insert in table
get $Tbl(x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k)$ suchthat M in P else P'	get from table
let $(x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) = O[M_1, \ldots, M_l](N_1, \ldots, N_k)$ in P else P'	oracle call
let $x[\widetilde{i}] : T =$ loop $O[M_1, \ldots, M_l](N)$ in P else P'	loop

This language distinguishes oracle declarations and oracle bodies. An oracle declaration provides some oracles, which can be called by the adversary, while an oracle body specifies the computations to perform upon oracle call, and returns the result of the oracle.

The oracle declaration 0 is empty: it declares no oracle at all. The oracle declaration $Q \mid Q'$ is a parallel composition: it simultaneously provides the oracles declared in Q and those in Q'. These oracles can be called in any order by the adversary. The oracle declaration $\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q$ provides n copies of the oracles declared in Q, indexed by $i \in [1, n]$, where n is a parameter (an unspecified integer). This parameter is used by CryptoVerif to express the maximum probability of breaking the protocol, which typically depends on the number of calls to the various oracles. Finally, the oracle declaration $O[\tilde{i}](x_1[\tilde{i}] : T_1, \ldots, x_k[\tilde{i}] : T_k) := P$ declares the oracle O, taking arguments $x_1[\tilde{i}], \ldots, x_k[\tilde{i}]$ of types $T_1, \ldots, T_k$ respectively. The result of this oracle is computed by the oracle body P. Similarly to variables, the oracle declaration declares an array of oracles, in order to differentiate multiple copies of an oracle.

The oracle body $x[\tilde{i}] \stackrel{R}{\leftarrow} T; P$ chooses a new random number uniformly in T, stores it in $x[\tilde{i}]$, and executes P. The type T must be a *fixed-length* type, which is a type that only contains all bitstrings of a certain length, because probabilistic Turing machines can choose random numbers uniformly only in such types. Function symbols represent deterministic functions, so all random numbers must be chosen by $x[\tilde{i}] \stackrel{R}{\leftarrow} T$. Using deterministic functions facilitates the proofs of protocols in CryptoVerif by making automatic syntactic manipulations easier: we can duplicate a term without changing its value. The assignment $x[\tilde{i}] \leftarrow M; P$ stores the value of M in x and executes P. The test $\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P'$ executes P when M evaluates to true and P' otherwise. The construct $\mathsf{event}\ e(M_1, \ldots, M_l); P$ executes the event $e(M_1, \ldots, M_l)$, then runs P. This event records that a certain program point has been reached with certain values of $M_1, \ldots, M_l$, but otherwise does not affect the execution of the system. (Events serve in specifying authentication properties [15].) The construct $\mathsf{return}(M_1, \ldots, M_k); Q$ returns the result $M_1, \ldots, M_k$ of the oracle. Additionally, it makes available the oracles defined in Q; these oracles can then be called by the adversary. The construct end terminates the oracle with an error, yielding control to the adversary.

The constructs insert and get handle tables, used for instance to store the keys of the protocol participants. A table can be represented as a list of tuples; $\mathsf{insert}\ Tbl(M_1, \ldots, M_k); P$ inserts the element $(M_1, \ldots, M_k)$ in the table Tbl; $\mathsf{get}\ Tbl(x_1[\tilde{i}] : T_1, \ldots, x_k[\tilde{i}] : T_k)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$ tries to retrieve an element $(x_1[\tilde{i}], \ldots, x_k[\tilde{i}])$ in the table Tbl such that M is true. When such an element is found, it executes P with $x_1[\tilde{i}], \ldots, x_k[\tilde{i}]$ bound to that element. When several such elements are found, one of them is chosen randomly. We cannot for instance take the first element found because the game transformations made by CryptoVerif may reorder the elements. For these transformations to preserve the behavior of the game, the distribution of the chosen element must be invariant by reordering. We cannot choose uniformly this element because there may not be exactly a power of two elements satisfying the property, but we choose using

an approximation of an uniform distribution that is good enough depending on the security parameters. Otherwise, when no such element is found, P' is executed.

The oracle call and loop constructs cannot be used in the protocol description Q, but are used to permit the attacker of the protocol, which is modelized by a process in this language in parallel with Q, to execute oracles at its disposition.

An oracle call $\mathsf{let}\ (x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) = O[M_1, \ldots, M_l](M'_1, \ldots, M'_k)\ \mathsf{in}\ P\ \mathsf{else}\ P'$ calls oracle $O[M_1, \ldots, M_l]$, stores its returned values in the variables $x_1[\widetilde{i}], \ldots, x_{k'}[\widetilde{i}]$, and continues with P if the oracle terminates with a return statement, and continues with P' if the oracle terminates with end.

A loop $\mathsf{let}\ x[\widetilde{i}] : T = \mathsf{loop}\ O[M_1, \ldots, M_n](M')\ \mathsf{in}\ P\ \mathsf{else}\ P'$ calls oracle O in a loop. Oracle O takes a unique argument (the internal state of the loop) and returns a pair containing a result of the same type (the modified internal state) and a boolean indicating whether the loop should continue or not. $O[M_1, \ldots, M_n](M')$ is first called. If it returns (a_1, true), $O[M_1 + 1, M_2, \ldots, M_n](a_1)$ is called. If it returns (a_2, true), $O[M_1 + 2, M_2, \ldots, M_n](a_2)$ is called, and so on, until $O[M_1 + k, M_2, \ldots, M_n](a_k)$ returns $(a_{k+1}, \mathrm{false})$. Then we run P with $x[\widetilde{i}]$ set to a_{k+1}. If O terminates with end, we run P'.

CryptoVerif also offers a pattern-matching construct. A function $f : T_1 \times \ldots \times T_m \to T$ that can be used for pattern-matching is declared with the attribute `compos`. This attribute means that f is injective and that its inverses are efficiently computable, that is, there exist efficiently computable functions $f_j^{-1} : T \to T_j$ $(1 \leq j \leq m)$ such that $f_j^{-1}(f(x_1, \ldots, x_m)) = x_j$. We can then define the pattern-matching construct $\mathsf{let}\ f(x_1, \ldots, x_m) = M\ \mathsf{in}\ P\ \mathsf{else}\ P'$ as an abbreviation for $y \leftarrow M; x_1 \leftarrow f_1^{-1}(y); \ldots; x_m \leftarrow f_m^{-1}(y); \mathsf{if}\ f(x_1, \ldots, x_m) = y\ \mathsf{then}\ P\ \mathsf{else}\ P'$. This construct tries to extract the values of $x_1, \ldots, x_n$ such that $f(x_1, \ldots, x_n) = M$, and runs P when this extraction succeeds, and P' when it fails. Also, we define the construct $\mathsf{let}\ (=M) = M'\ \mathsf{in}\ P\ \mathsf{else}\ P'$ as $\mathsf{if}\ \ M = M'\ \mathsf{then}\ P\ \mathsf{else}\ P'$. We generalize this construct to $\mathsf{let}\ pat = M\ \mathsf{in}\ P\ \mathsf{else}\ P'$ where pat is built from `compos` functions, variable names, and equality to terms $=M$.

Example 1 Let $f : T_1 \times T_2 \to T$ be a function with the `compos` attribute. Let $f_1^{-1} : T \to T_1$ and $f_2^{-1} : T \to T_2$ be efficiently computable inverses of f.

The oracle body $\mathsf{let}\ f(=M, x) = M'\ \mathsf{in}\ P\ \mathsf{else}\ P'$ is an abbreviation of:

$$y \leftarrow M'; x_1 \leftarrow f_1^{-1}(y); x \leftarrow f_2^{-1}(y); \mathsf{if}\ f(x_1, x) = y \wedge M = x_1\ \mathsf{then}\ P\ \mathsf{else}\ P'\,,$$

where y is a fresh variable name. If there exists a value x such that $f(M, x) = M'$, it runs P with that value of x, else it runs P'. □

An else branch of if, get, or let may be omitted when it is else end. Similarly, end may be omitted after a random choice, an assignment, an event, or a table insertion. A trailing 0 after a return may also be omitted.

Example 2 Let us consider a simple protocol in which the first participant A generates a nonce x, and sends it to the second participant B encrypted under the shared secret key K_{ab}: $A \to B : \{x\}_{K_{ab}}$. This protocol can be modeled in

CryptoVerif as follows:

$$\begin{array}{l}\text{Okeygen}() := rK_{ab} \stackrel{R}{\leftarrow} keyseed;\ K_{ab} \leftarrow \text{kgen}(rK_{ab}); \mathsf{return}();\\ \quad (\mathsf{foreach}\ i_1 \leq n_1\ \mathsf{do}\ P_A \mid \mathsf{foreach}\ i_2 \leq n_2\ \mathsf{do}\ P_B)\end{array}$$

$$P_A = \text{OA}() := x \stackrel{R}{\leftarrow} nonce;\ s \stackrel{R}{\leftarrow} seed; \mathsf{return}(\text{enc}(x, K_{ab}, s))$$

$$P_B = \text{OB}(m : bitstring) := \mathsf{let}\ \text{injbot}(r') = \text{dec}(m, K_{ab})\ \mathsf{in}\ \mathsf{return}()$$

The only oracle callable at the beginning is Okeygen, which generates a symmetric encryption key K_{ab} by generating a random seed rK_{ab} and using the key generation algorithm kgen on it. It returns nothing. The key K_{ab} is available to the following oracles in the process, but is not given to the adversary. After having called Okeygen, one can call n_1 times the oracle OA and n_2 times the oracle OB. In the oracle OA, we generate a nonce x, a seed for the encryption s, and return the encryption of x under the key K_{ab} with the random seed s. The oracle OB takes as argument m, which should be the message returned by the oracle OA. It decrypts the message under the symmetric key K_{ab}. A decrypted message is of type $bitstring_\perp$: it can be a bitstring or the $\perp$ value, which means that decryption failed. The function injbot is the injection that takes a *nonce* value and returns its value in $bitstring_\perp$, which is different from $\perp$. When decryption succeeds, the oracle OB stores in r' the result of the decryption, and returns normally. Otherwise, it terminates with end (implicit in the omitted else branch of let).

Adversaries are processes in parallel with the protocol process defined above. The following process is an adversary for the previously defined protocol:

$$\begin{array}{l}O_{\text{start}}() :=\\ \quad \mathsf{let}\ () = \text{Okeygen}[\,]()\ \mathsf{in}\\ \quad \mathsf{let}\ (m[\,] : bitstring) = \text{OA}[1]()\ \mathsf{in}\\ \quad \mathsf{let}\ () = \text{OB}[1](m)\ \mathsf{in}\\ \quad \mathsf{return}()\end{array}$$

This process defines the oracle O_{start}, which is the entry point of CryptoVerif. It begins by calling the key generation oracle Okeygen, then calls OA, stores the message it received from it in m, and finally calls OB with this message, and returns or ends similarly to OB. This represents a normal run of the protocol.

An adversary can only call an oracle if it is available, the adversary process

$$O_{\text{start}}() := \mathsf{let}\ (m[\,] : bitstring) = \text{OA}[1]()\ \mathsf{in}\ \mathsf{return}()$$

blocks when trying to call OA. □

CryptoVerif verifies the following requirements on a process:

Property 1.1 *Variables are renamed so that each variable has a single definition. The indices $\widetilde{i}$ of a variable $x[\widetilde{i}]$ are always the indices of replications above the definition of x.*

Property 1.2 *The processes are well-typed. (In particular, functions and oracles receive arguments of their expected types. For brevity, we do not detail the type system; see [16] for a similar type system.)*

Property 1.3 *Oracles with the same name can be defined only in different branches of an* if *or* get *construct. In an oracle definition* $O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P$*, the indices* $\widetilde{i}$ *are always the indices of replications above that oracle definition.*

Property 1.4 *We define types of oracles as follows. The type of a* return$(M_1, \ldots, M_k); Q$ *statement consists of the types of* $M_1, \ldots, M_k$ *and the list of types of the oracle definitions at the beginning of* Q*, ordered from left to right. The type of an oracle definition consists of the oracle name, the bounds of the replications above that oracle definition, the types of the arguments of the oracle, and the common type of its return statements.*

An oracle may have several return *statements, but they must be of the same type. When there are several definitions of an oracle with the same name* O*, they must be of the same type.*

Property 1.1 makes sure that a distinct array cell is used in each copy of a process, so that all values of the variables during execution are kept in memory. (This helps in cryptographic proofs.) To lighten notations, we often omit the indices since they are determined by Property 1.1. Property 1.2 requires the adversary to be well-typed. This requirement does not restrict its computing power, because it can always define type-cast functions $f : T \to T'$ to bypass the type system. Similarly, the type system does not restrict the class of protocols that we consider, since the protocol may contain type-cast functions. The type system just makes explicit which set of bitstrings may appear at each point of the protocol. Property 1.4 guarantees that the various definitions of an oracle are consistent, and can in fact be compiled into a single function in OCaml. Property 1.3 guarantees that there exists a single callable definition for each oracle.

1.3 Annotations for Implementation

The protocol specification language also includes annotations to specify which parts of the protocol will be compiled into which OCaml modules, and which OCaml types, functions, and files correspond to the CryptoVerif types, functions, and key tables. These annotations are simply ignored when CryptoVerif proves the protocol.

A protocol typically includes several parts of code run by different participants, for instance a client and a server. These parts of code will be included in different programs, so we split the protocol into multiple roles role that will be translated into different OCaml modules. The boundaries of roles are marked as follows. The annotation role$[x_1 >$ "*filex*$_1$", $\ldots, x_n >$ "*filex*$_n$", $y_1 <$ "*filey*$_1$", $\ldots, y_m <$ "*filey*$_m$"$]\{$ indicates the beginning of the role role. It should be placed just above an oracle declaration Q. The indication $x_i >$ "*filex*$_i$" means that the variable x_i

will be stored in file filex_i when it is defined. The variable x_i can then be used in other roles defined after the end of role; these roles will read it automatically from the file filex_i. The indication $y_i <$ "filey_i" means that the role role will read at initialization the value of the variable y_i from the files filey_i. The variable y_i must be free in role (i.e., it is defined before the beginning of role). A declaration $x >$ "filex" in a role role$'$ above role implicitly implies $x <$ "filex" in role when role uses x: x is written to filex in role$'$ and read in role. All variables free in role role must be declared as being read from a file in role, either explicitly or implicitly as mentioned above. All variables read from or written to a file must be defined under no replication. (Otherwise, several copies of the variable would have to be stored in the file.) Storing variables in files is useful for variables that are communicated across roles, for example long-term keys that are set in a key generation program and later used by the client and/or server programs. The closing brace } indicates the end of the current role. It must be placed just after a return statement.

Example 3 Let us annotate the process we have seen in Example 2.

$$\begin{aligned}
&\mathsf{role}_{Keygen}[K_{ab} > \text{"}\mathit{keyfile}\text{"}]\{\mathrm{Okeygen}() := \ldots \mathsf{return}()\};\\
&\qquad (\mathsf{foreach}\ i_1 \leq n_1\ \mathsf{do}\ P_A \mid \mathsf{foreach}\ i_2 \leq n_2\ \mathsf{do}\ P_B)\\
&P_A = \mathsf{role}_A\{\mathrm{OA}() := \ldots\\
&P_B = \mathsf{role}_B\{\mathrm{OB}(m : \mathit{bitstring}) := \ldots
\end{aligned}$$

We divide the process into three roles. First, the key generation role is represented by role_{Keygen}, containing just the oracle Okeygen. We store the value of K_{ab} in the file *keyfile*, in order to be able to read the value of the key in the other roles. The role role_A, which contains the oracle OA, corresponds to the role of A, and the role role_B, which contains the oracle OB, corresponds to the role of B. For these two roles, there is no need to write the closing brace } because there is nothing after them. □

The correspondence between CryptoVerif and OCaml types, functions, and tables is specified by declarations in the input file. These declarations associate to each CryptoVerif type T:

- its corresponding OCaml type $\mathbb{G}_T(T)$.
- the serialization function $\mathbb{G}_{\mathsf{ser}}(T)$ of type $\mathbb{G}_T(T) \to \mathsf{string}$, which converts an element of type $\mathbb{G}_T(T)$ to a bitstring, and the deserialization function $\mathbb{G}_{\mathsf{deser}}(T)$ of type $\mathsf{string} \to \mathbb{G}_T(T)$, which performs the inverse operation. These functions serve for writing values to files and for reading them. When deserialization fails, it must raise the exception Bad_file; this exception is raised only when a file has been corrupted.
- the predicate function $\mathbb{G}_{\mathsf{pred}}(T)$ of type $\mathbb{G}_T(T) \to \mathsf{bool}$, which returns whether an OCaml element of type $\mathbb{G}_T(T)$ belongs to type T or not. Indeed, the CryptoVerif values of type T may correspond only to a subset of the OCaml values of type $\mathbb{G}_T(T)$.

- the random number generation function $\mathbb{G}_{\mathsf{random}}(T)$, of type $\mathsf{unit} \to \mathbb{G}_T(T)$, which returns a random element uniformly chosen in type T.

These declarations actually need only to be made if the corresponding feature is used in the protocol.

The user must also declare, for each free variable $x[\,]$ in a role, the file f in which the variable will be stored. Let files be the set of these pairs $(x[\,], f)$. Let also tables be the set of pairs (Tbl, f) such that the table Tbl will be stored in file f. Each CryptoVerif function f of type $T_1 \times \cdots \times T_m \to T$ must associate the corresponding OCaml function $\mathbb{G}_{\mathsf{f}}(f)$ of type $\mathbb{G}_T(T_1) \times \cdots \times \mathbb{G}_T(T_m) \to \mathbb{G}_T(T)$.

These correspondences are specified in the specification by the following implementation declarations in the CryptoVerif input file:

- implementation type $T = \mathbb{G}_T(T)$ [*options*] sets the OCaml type corresponding to the CryptoVerif type T. The possible options are:
  - *serial* $= \mathbb{G}_{\mathsf{ser}}(T), \mathbb{G}_{\mathsf{deser}}(T)$ sets the serialization and deserialization functions for type T.
  - *pred* $= \mathbb{G}_{\mathsf{pred}}(T)$ sets the predicate function.
  - *random* $= \mathbb{G}_{\mathsf{random}}(T)$ sets the random number generation function.
- implementation type $T = n$ [*options*] sets the length of the type T. The type T is then represented by a bitstring of length n. If n is a multiple of 8, then T will be represented by a string: $\mathbb{G}_T(T) = \mathsf{string}$; if $n = 1$, then T will be represented by a boolean: $\mathbb{G}_T(T) = \mathit{bool}$. Otherwise, an error occurs. The only allowed option is *serial*, which allows one to override the default serialization and deserialization functions to choose a different representation of the bitstring. This sets the functions $\mathbb{G}_{\mathsf{random}}(T)$ and $\mathbb{G}_{\mathsf{pred}}(T)$ to correct values.
- implementation table Tbl = *file* sets the file in which the table Tbl is written.
- implementation fun $f = \mathbb{G}_{\mathsf{f}}(f)$ [*options*] sets the translation of the function f. If f has the `compos` attribute, that is to say that f is injective, this declaration can take the option *inverse* $= \mathbb{G}_{\mathsf{inv}}(f)$, which declares $\mathbb{G}_{\mathsf{inv}}(f)$ as the inverse function. If f is of type $T_1 \times \ldots \times T_m \to T$, this function must be of type $\mathbb{G}_T(T) \to \mathbb{G}_T(T_1) \times \ldots \times \mathbb{G}_T(T_m)$. $\mathbb{G}_{\mathsf{inv}}(f)\ x$ must return a tuple $(x_1, \ldots, x_m)$ such that $\mathbb{G}_{\mathsf{f}}(f)\ x_1\ \ldots\ x_m = x$. If there is no such element, $\mathbb{G}_{\mathsf{inv}}(f)$ must raise Match_failure. The function $\mathbb{G}_{\mathsf{inv}}(f)$ is used for translating the pattern-matching construct into OCaml; for simplicity, we do not detail this translation.
- implementation const $f = \mathbb{G}_{\mathsf{f}}(f)$ sets the implementation of the function f that has no arguments to the OCaml constant $\mathbb{G}_{\mathsf{f}}(f)$.

Example 4 For instance, the declaration

implementation type *nonce* = "string" [*serial* = "`id`", "`deserial 64`";
pred = "`sizep 64`";
random = "`rand_string 8`"].

means that the CryptoVerif type *nonce* is implemented by the OCaml type string, with:

- serialization function identity,
- deserialization function `deserial 64` which is the identity for strings of 8 bytes (64 bits) and raises Match_failure for other strings,
- predicate function `sizep 64` which returns true for strings of 8 bytes and false for other strings,
- and random number generation function `rand_string 8` which returns random strings of 8 bytes.

In other words, nonces are bitstrings of 64 bits, which we can abbreviate by

implementation type *nonce* = 64.

The declaration

implementation fun kgen = "kgen".

means that the CryptoVerif function kgen is implemented by the OCaml function `kgen`. □

CryptoVerif verifies the following properties on an instrumented process:

Property 1.5 *There is a single occurrence of each role* role*. If an oracle O has a* return$(x_1, \ldots, x_k)\};Q$ *where Q contains a role definition, then there is only one* return *statement for the oracle O in the whole initial process.*

This property guarantees that we know which process to compile for a given role, and which roles start after the return from a given oracle.

Property 1.6 *Roles cannot be nested.*

1.4 Improvements on Syntax

1.4.1 Tables

The original CryptoVerif language does not include insert and get. Instead, it offers a construct for looking up values in arrays, find. The constructs insert and get are intuitively easier to understand, closer to the constructs used by cryptographers, and much easier to implement. However, arrays and find are very helpful for the automatic proofs performed by CryptoVerif, as explained in [16]. Therefore, in order to implement insert and get, we first transform them into arrays and find, so that CryptoVerif can run as before after this transformation.

Originally, terms have the following syntax:

$M, N ::=$	terms
a	bitstring
i	index
$x[M_1, \ldots, M_m]$	variable
$f(M_1, \ldots, M_m)$	function application

Variables are of the form $x[M_1, \ldots, M_m]$: we can access all values of a variable and not only the current one. The tuple $M_1, \ldots, M_m$ corresponds to the indices of the replications above the definition of x: they indicate which value of x we access.

The find construct has the following syntax:

$$\mathsf{find}\left(\bigoplus_{j=1}^{m} \begin{array}{l} u_{j1} \leq n_{j1}, \ldots, u_{jm_j} \leq n_{jm_j} \ \mathsf{suchthat} \\ \mathsf{defined}(M_{j1}, \ldots, M_{jl_j}) \wedge M_j \ \mathsf{then}\ P_j \end{array}\right) \mathsf{else}\ P$$

This construct finds indices $u_{j1}, \ldots, u_{jm_j}$ such that $M_{j1}, \ldots, M_{jl_j}$ are defined and M_j is true. If such indices are found, it runs P_j. If no such indices can be found for any j, it runs P. More formally, this construct computes the set S of elements $j, a_1, \ldots, a_{m_j}$ where $a_1, \ldots, a_{m_j}$ are replication indices such that all terms M_{jk} are defined and M_j evaluates to true, after replacing each u_{jk} with a_k. If the set S is empty, no instance of the replication indices could satisfy the conditions and so we continue with P. Otherwise, we choose randomly (with an almost uniform distribution) an element in S, and if the chosen element is $j_0, a_1, \ldots, a_{m_{j_0}}$ we instantiate the variables $u_{j_0 1}, \ldots, u_{j_0 m_{j_0}}$ to $a_1, \ldots, a_{m_{j_0}}$ and continue with P_{j_0}.

Example 5 Let us show how this construct can be used. Let us consider the following process:

$$\begin{array}{l} \quad \mathsf{foreach}\ i_1 \leq n_1\ \mathsf{do}\ \mathrm{Insert}[i_1](x[i_1] : id, k[i_1] : key) := \mathsf{end} \\ \mid \mathsf{foreach}\ i_2 \leq n_2\ \mathsf{do}\ \mathrm{Get}[i_2](x'[i_2] : id) := \\ \qquad \mathsf{find}\ u \leq n_1\ \mathsf{suchthat}\ \mathsf{defined}(x[u], k[u]) \wedge x[u] = x'[i_2]\ \mathsf{then}\ \mathsf{return}(k[u]) \\ \qquad \mathsf{else}\ \mathsf{end} \end{array}$$

This process defines two oracles, Insert and Get. The goal of these oracles is to implement an association table that associates identifiers to keys. The oracle Insert adds an identifier and a key in the table: when one calls Insert, we remember the values given as argument inside the arrays x and k. The oracle Get is used to retrieve the key corresponding to an identifier: when called, we search an index $u \leq n_1$ such that:

- $x[u]$ and $k[u]$ are defined, that is to say, an index u such that the oracle Insert has been called with replication index u,
- and the identifier $x[u]$ is equal to the identifier given as argument to Get.

So, the oracle Get returns either the key $k[u]$, which is one of the keys associated to the requested identifier, or ends if no key is registered for the requested identifier.

An equivalent process using insert and get would be:

$$\begin{array}{l} \quad \mathsf{foreach}\ i_1 \leq n_1\ \mathsf{do}\ \mathrm{Insert}'[i_1](x[i_1] : id, k[i_1] : key) := \mathsf{insert}\ Tbl(x[i_1], k[i_1]) \\ \mid \mathsf{foreach}\ i_2 \leq n_2\ \mathsf{do}\ \mathrm{Get}'[i_2](x'[i_2] : id) := \\ \qquad \mathsf{get}\ Tbl(x''[i_2], k''[i_2])\ \mathsf{suchthat}\ x''[i_2] = x'[i_2]\ \mathsf{in}\ \mathsf{return}(k''[i_2])\ \mathsf{else}\ \mathsf{end} \end{array}$$

Oracle Insert′ inserts into the table Tbl the identifier and key we pass to it, and oracle Get′ finds a pair $(x''[i_2], k''[i_2])$ in table Tbl such that the identifier $x''[i_2]$ is the identifier $x'[i_2]$ given in argument. □

The transformation of insert and get into find proceeds by storing the inserted list elements in fresh array variables, and looking up in these arrays instead of performing get. More precisely, when insert $Tbl(M_1, \ldots, M_k); P$ is under the replications foreach $i_l \leq n_l$ do ... foreach $i_1 \leq n_1$ do, it is transformed into

$$y_1[i_1, \ldots, i_l] \leftarrow M_1; \ldots; y_k[i_1, \ldots, i_l] \leftarrow M_k; P$$

where $y_1, \ldots, y_k$ are fresh array variables, and we add the tuple

$$(y_1, \ldots, y_k; i_1 \leq n_1, \ldots, i_l \leq n_l)$$

in a set S', to remember them. The construct

$$\mathsf{get}\ Tbl(x_1 : T_1, \ldots, x_k : T_k)\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P'$$

is then transformed into

$$\mathsf{find}\left(\bigoplus_{(y_1,\ldots,y_k;i_1\leq n_1,\ldots,i_l\leq n_l)\in S'} \begin{array}{l} u_1 \leq n_1, \ldots, u_l \leq n_l\ \mathsf{suchthat} \\ \mathsf{defined}(y_1[\widetilde{u}], \ldots, y_k[\widetilde{u}]) \wedge \\ M\{y_1[\widetilde{u}]/x_1, \ldots, y_k[\widetilde{u}]/x_k\} \\ \mathsf{then}\ x_1 \leftarrow y_1[\widetilde{u}]; \ldots;\ x_k \leftarrow y_k[\widetilde{u}];\ P \end{array}\right)$$
$$\mathsf{else}\ P'$$

where $\widetilde{u}$ stands for $u_1, \ldots, u_l$. This construct looks in all arrays used for translating insertion in table Tbl, for indices $\widetilde{u}$ such that $y_1[\widetilde{u}], \ldots, y_k[\widetilde{u}]$ are defined, that is, an element has been inserted at indices $\widetilde{u}$, and $M\{y_1[\widetilde{u}]/x_1, \ldots, y_k[\widetilde{u}]/x_k\}$ is true, that is, that element satisfies M. When it finds such an element, it stores it in $x_1, \ldots x_k$, and runs P. (When it finds several elements, one of them is chosen randomly with the same approximation of a uniform distribution as in get.) When it finds no element, it executes P'.

The defined predicate can also be used on tests: the construct

$$\mathsf{if}\ \mathsf{defined}(M_1, \ldots, M_j) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'$$

tests whether the terms $M_1, \ldots, M_j$ are defined and M is true, and in this case, it runs P. Otherwise, it runs P'. It is an abbreviation of

$$\mathsf{find}\ \mathsf{suchthat}\ \mathsf{defined}(M_1, \ldots, M_j) \wedge M\ \mathsf{then}\ P\ \mathsf{else}\ P'\,.$$

1.4.2 Function Macros

A trick can be used to provide, for the same function f, both an OCaml implementation and a CryptoVerif definition of f from other functions. Indeed, CryptoVerif allows one to define f as a macro: letfun $f(x_1 : T_1, \ldots, x_m : T_m) = M$. Specifying an OCaml implementation for these macros is optional. When the OCaml implementation is not specified, our compiler generates code according to the letfun macro. When the OCaml implementation is specified, it is used for generating the OCaml code, while the CryptoVerif macro defined by letfun is used for proving the protocol. This feature can be used, for instance, to define probabilistic functions: the OCaml implementation generates the random

choices inside the function, while the CryptoVerif definition by letfun first makes the random choices, then calls a deterministic function.

We added this functionality in order to use cryptographic libraries that embed the random number generation in their primitives.

Example 6 We can define an encryption function that generates the random seed internally as follows:

$$\mathsf{letfun}\ \mathrm{renc}(x : \mathit{bitstring}, k : \mathit{key}) = s \xleftarrow{R} \mathit{seed};\ \mathrm{enc}(x, k, s).$$

where enc is a deterministic encryption function that takes the random seed as argument. We can give an OCaml implementation for renc by

$$\mathsf{implementation\ fun}\ \mathrm{renc} = \texttt{"renc"}.$$

Obviously, this OCaml function must also choose the random seed for encryption internally. □

1.5 New Game Transformations

In this section, we present our modifications to CryptoVerif strategies to prove correctness of protocols.

1.5.1 Fact Collection

CryptoVerif defines the transformation replace that allows the user to transform a term in the game into another. The tool must be able to prove that the new game where this term has been replaced is distinguishable from the game where the replacement has not been done only with a negligible probability, otherwise the transformation will fail.

In order to prove this, the tool collects facts: for example, if we are in the then branch of an if M then P else P' construct, the term M is true at this point. We improved this fact collection so that, if the term we want to replace is inside a conjunction or a disjunction, we take into account other parts of this conjunction or disjunction. Suppose we want to replace the term M in $M \wedge M_1$, we can suppose that M_1 is true: if the term M_1 is false, the complete term $M \wedge M_1$ is also false, and M can be replaced by any value in this case. So, we can add the fact that M_1 is true to the collection of facts. By the same reasoning, when considering the replacement of the term M in $M \vee M_2$, we add the fact that M_2 is false to the collection of facts.

After applying a game transformation, CryptoVerif tries to simplify the game. This simplification also uses the facts we collect on the process. We also adapted this technique so that simplification can take advantage of facts derived from conjunctions and disjunctions.

In particular, this technique permits us to simplify terms like $M \wedge M$, which is first simplified to $M \wedge \mathrm{true}$, and then to M. One must take care not to simplify more than one part of a conjunction or disjunction at once, otherwise the simplification becomes unsound: the term $M \wedge M$ would be transformed into $\mathrm{true} \wedge \mathrm{true}$!

1.5.2 Case Distinction on Variable Creation Order

In our proof of SSH, we needed the game transformation described thereafter.

Let us begin by explaining the transformation on an example. Consider the game of Figure 1.3.

Figure 1.3 Example game to present the extension

$$
\begin{array}{l}
\mathsf{foreach}\ i \leq n\ \mathsf{do}\ \mathrm{A}(a : T) := \\
\quad \mathsf{find}\ v \leq N'\ \mathsf{suchthat}\ \mathsf{defined}(b[v]) \wedge b[v] = a\ \mathsf{then\ end} \\
\quad \mathsf{else}\ k \leftarrow a \\
\\
\mathsf{foreach}\ j \leq n'\ \mathsf{do}\ \mathrm{B}() := \\
\quad b \stackrel{R}{\leftarrow} T; \\
\quad \mathsf{return}(b); \\
\\
\quad \mathrm{B}'() := \\
\quad \mathsf{find}\ w \leq N\ \mathsf{suchthat}\ \mathsf{defined}(a[w]) \wedge b = a[w]\ \mathsf{then} \\
\quad\quad \mathsf{if}\ \mathsf{defined}(k[w])\ \mathsf{then}\ P
\end{array}
$$

The type T must be a *large* type: the type contains enough elements so that the probability of collision between a random element in T and an independent value in T is negligible.

The first oracle A takes as argument the variable a of type T and searches whether there is an occurrence of the value a in all the values that were taken by the variable b, created randomly by the second oracle. If there were none, we create a variable k. The second oracle B creates a new random value b of type T. The last oracle B′ searches whether there is a run w of oracle A where the contents of variable $a[w]$ is equal to the contents of b. In this case, we test whether the variable $k[w]$ is defined.

To give some intuition on the game, the oracles present in the game model honest participants to a key exchange protocol. The variable b represents a key created by B. Oracle A looks up whether the given key in argument is a key generated by oracle B, and creates the variable k when this is not the case. The existence of variable $k[w]$ represents the fact that the adversary attempted to forge the message intended to the wth run of A. The process P is run when the adversary successfully forged a value $a[w]$, the argument given to the wth run of oracle A, such that b, the key generated by B, is equal to this value.

We want to prove that the program P is run only with negligible probability. Let us suppose that we are in the then branch of the if of oracle B′. The variable $k[w]$ is defined. So the oracle A has been run with replication index w, and in this run, the condition of the find of oracle A was false at that point. So, for all v, $\mathsf{defined}(b[v]) \wedge b[v] = a[w]$ did not hold when we ran the wth run of oracle A.

We have two cases.

- If the variable $k[w]$ is defined after $b[j]$, then the variable $a[w]$ is also defined after $b[j]$, because there is no parallelism in CryptoVerif, the oracles are called one after another. The variable $b[j]$ is then defined when the oracle A runs with replication index w. So by the above property, by taking $v = j$, we have that $\mathsf{defined}(b[j]) \wedge b[j] = a[w]$ does not hold, so $b[j] \neq a[w]$. The condition of the find construct of oracle B′ is not satisfied, so there is a contradiction: we cannot access the then branch of the if statement.

- Otherwise, the variable $k[w]$ is defined before $b[j]$. The variable $b[j]$ is chosen randomly after the assignment of $a[w]$. So the variable $b[j]$ is independent of $a[w]$. Therefore, the probability that $b[j] = a[w]$ is the probability of collision of a random value of type T with $a[w]$, this probability is negligible because T is a large type. We also cannot access the then branch of the if statement, with overwhelming probability.

So we can replace the if statement by the contents of its else branch, that is to say the end process, with overwhelming probability.

The algorithm of the transformation consists of the following steps that we apply on each find or if branch B. Let us present the algorithm on the then branch of the if of oracle B′.

1. We first search whether a variable $k[w]$ is defined in the branch B such that this variable is defined in an else branch of a find F. The find F is the find of oracle A.

2. Facts coming from an else branch of a find have all the form

$$\forall u_1 \leq n_1, \ldots, u_k \leq n_k, \neg(\mathsf{defined}(M_1, \ldots, M_k) \wedge M)\,,$$

   which indicate that no candidate $(u_1, \ldots, u_k)$ satisfied the condition

$$u_1 \leq n_1, \ldots u_k \leq n_k \text{ suchthat } \mathsf{defined}(M_1, \ldots, M_k) \wedge M$$

   of a branch of the find.

   We try to adapt these facts by choosing intelligent values for $u_1, \ldots, u_k$. To do that, we try to transform the indices $u_1, \ldots, u_k$ so that variables present in terms $M_1, \ldots, M_k, M$ correspond to variables present near branch B.

   The fact we obtain from F is $\forall v \leq n', \neg(\mathsf{defined}(b[v]) \wedge b[v] = a[w])$. Variable $b[j]$ is used near B, so we substitute the index v by j, and we get the fact $\neg(\mathsf{defined}(b[j]) \wedge b[j] = a[w])$.

3. If CryptoVerif is able to prove that the true facts at point B and this new fact contradict, we have proven the first point: when we get to the branch B before the call to the find F, we cannot get to branch B.

4. We then try to find two variables that correspond to variables $k[w]$ and $b[j]$ on which we distinguish cases.

   The first variable k must be defined in the else branch of F.

The second variable b must be defined structurally before the branch B and after the first variable. The variable b must be a variable whose contents are chosen at random from a large type.

We choose the variables this way to ensure that if $k[w]$ is defined after $b[j]$ temporally, then the wth run of the find F is also defined temporally after the jth run of branch B.

5. For each pair of variables $(k[w], b[j])$ we obtain in the previous step, we simplify the game, given the assumption that the second variable is independent of the first. If after simplification (that considers collisions), we also get a contradiction, we have proven that the branch B is inaccessible.

Chapter 2

OCaml

This chapter presents the OCaml language, the target language of our compiler.

OCaml is a functional language that has strong typing, type inference, and has features allowing one to code in an imperative, functional, or object oriented style.

We only use a small core of the language in our translation, and we will present only the part of the language we use.

Figures 2.1 and 2.2 summarize the syntax of our subset of OCaml. OCaml expressions e, when evaluated, return OCaml values v, presented in Figure 2.3, or exceptional values raise v. Let us describe the language by explaining the syntax constructs. For brevity, we ignore types in this syntax.

Pattern-matching. Pattern-matching is a central feature of OCaml. A pattern *pat* describes the form of a value to be matched. When we match a value v with a pattern *pat*, if the value is of the correct form, then we bind each variable x occurring in the pattern *pat* to the corresponding part of v. Patterns must be linear, that is, no variable can occur more than once inside a pattern. The special pattern _ matches any value, without binding the matched value to any variable, and there can be any number of these in a pattern. When we match a value v with the pattern matching $pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n$, we match v sequentially to the patterns $pat_1, \ldots, pat_n$. If the first pattern that matches v is pat_i, then we evaluate e_i. If no pattern matches v, then we raise the exception Match_failure.

Exceptions. Exceptions are used as a mechanism of error passing in the language. We denote the expressions of the form raise v which are the applications of the primitive raise to exception values v *exceptional values*.

All constructs of the language except try return directly the exceptional value raise v when an expression e of the construct is this exceptional value. The construct try raise v with pm matches the exception v in the pattern-matching pm. So the construct try e with pm is used to catch exceptions that can be raised inside the expression e. For example, the expression try let $x =$ raise v in e with $v \to e'$ evaluates e'. When the exception is not found in the pattern-matching, this raises again the exception raise v.

Figure 2.1 OCaml expressions

$pat ::=$	pattern
$\quad x$	$\quad$ variable
$\quad \_$	$\quad$ universal pattern
$\quad (pat_1, \ldots, pat_n)$	$\quad$ tuple
$\quad pat_1 :: pat_2$	$\quad$ list constructor
$pm ::=$	pattern matching
$\quad pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n$	$\quad$ pattern matching
$e ::=$	expression
$\quad prim$	$\quad$ primitive
$\quad x$	$\quad$ variable
$\quad l$	$\quad$ location
$\quad c$	$\quad$ constant ([], (), 0, false, ...)
$\quad (e_1, \ldots, e_n)$	$\quad$ tuple
$\quad e_1 :: e_2$	$\quad$ list constructor
$\quad \mathsf{function}\ pm$	$\quad$ function
$\quad e_1\ e_2$	$\quad$ application
$\quad e_1; e_2$	$\quad$ sequence
$\quad \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	$\quad$ if
$\quad \mathsf{match}\ e\ \mathsf{with}\ pm$	$\quad$ pattern matching
$\quad \mathsf{try}\ e\ \mathsf{with}\ pm$	$\quad$ try
$\quad \mathsf{let}\ pat = e_1\ \mathsf{in}\ e_2$	$\quad$ let
$\quad \mathsf{let\ rec}\ x_1 = \mathsf{function}\ pm_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ x_n = \mathsf{function}\ pm_n\ \mathsf{in}\ e$	$\quad$ let rec
$\quad \mathsf{function}[env, pm]$	$\quad$ closure
$\quad \mathsf{letrec}[env, \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}\ \mathsf{in}\ x_i]$	$\quad$ let rec closure, $1 \leq i \leq n$

Figure 2.2 OCaml programs

$d ::=$	definition
$\quad \mathsf{let}\ pat = e$	$\quad$ let
$\quad \mathsf{let\ rec}\ x_1 = \mathsf{function}\ pm_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ x_n = \mathsf{function}\ pm_n$	$\quad$ let rec
$definitions ::=$	definitions
$\quad \varepsilon$	$\quad$ empty definition list
$\quad d;;\ definitions$	$\quad$ definition list
$program ::=$	program
$\quad definitions$	$\quad$ list of definitions
$\quad \mathsf{raise}\ v$	$\quad$ exception

Figure 2.3 OCaml values

$v ::=$	value
$\mathit{prim}\ v_1\ \ldots\ v_j$	partially applied primitives (prim is n-ary and $0 \leq j < n$)
c	constant ([], (), 0, false, ...)
l	location
$(v_1, \ldots, v_n)$	tuple
$v_1 :: v_2$	list constructor
$\mathsf{function}[env, pm]$	closure
$\mathsf{letrec}[env, \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}\ \mathsf{in}\ x_i]$	let rec closure

Primitives. The basic operations of the language are implemented by primitives *prim*. We write binary primitives in infix notation: for example, we write $v_1 = v_2$ rather than $(=)\ v_1\ v_2$. We consider the following primitives: not is the boolean negation, $(=)$ is the equality test. The primitive raise e raises the exception e: it returns the exceptional value raise v if e reduces into v.

We use primitives to manage references, which are mutable memory cells. We represent memory cells by locations l. The reference creation ref v creates a new location l, stores the value v in l, and returns the location l. The assignment $l := v$ replaces the content of the location l with the value v. The dereference $!l$ returns the content of the location l.

The language also includes primitives to manage other native types such as integers (e.g., addition and multiplication) and strings (e.g., concatenation, extraction of substrings, and conversion between integers in $\{0, \ldots, 255\}$ and one-character strings). Strings are immutable values in our semantics. In contrast, in OCaml, values of type `string` are mutable. Our strings could be implemented in OCaml as an abstract type, on which only operations that do not mutate strings are implemented.

Expressions. Most expressions are standard. Locations cannot appear in the initial program. Constants c can be integers, strings, boolean values true or false, the empty list [], the unit constant (), and exceptions. The expression function *pm* defines a function. When this function is applied to a value v, it matches that value using the pattern matching *pm*. The application $e_1\ e_2$ applies the function e_1 to the argument e_2. The sequence operation $e_1; e_2$ evaluates e_1, ignoring its result (but obviously keeping its side effects), then evaluates e_2. The matching operation match e with *pm* evaluates e and matches the result of e using the pattern matching *pm*. The try construct try e with *pm* behaves as presented in Paragraph Exceptions. The let binding let $\mathit{pat} = e_1$ in e_2 evaluates e_1, matches the result with the pattern *pat*, which binds the variables in *pat*, and finally evaluates e_2. When the pattern matching fails, it raises the exception Match_failure. This construct is equivalent to match e_1 with $\mathit{pat} \to e_2$. The let rec binding let rec $x_1 =$ function pm_1 and ... and $x_n =$ function pm_n in e defines n mutually recursive functions $x_1, \ldots, x_n$, and evaluates the expression e using

these functions.

Closures are not present in the initial program, but they serve to represent functional values internally. The closure $\mathsf{function}[env, pm]$ comes from the function $\mathsf{function}\ pm$. It contains the code of the function (pm), and an environment env that maps the free variables of pm to their values. Closures allow one to evaluate functions using the values that the free variables of the function had at the definition of the function. (In other words, OCaml uses static variable binding.) The let rec closure $\mathsf{letrec}[env, \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}\ \mathsf{in}\ x_i]$ is similar, but for mutually recursive functions. It records all the mutually recursive bindings together.

We define the list expression $[e_1; e_2; \ldots; e_n]$ as syntactic sugar for $e_1 :: (e_2 :: \ldots :: (e_n :: [\,])\ldots)$. The expression $e \,\&\&\, e'$ is syntactic sugar for $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}$ false, and $e \,||\, e'$ is syntactic sugar for $\mathsf{if}\ e\ \mathsf{then}$ true $\mathsf{else}\ e'$.

Program. A program is a list of top level definitions d, or the raising of an exception, for exceptions that are not catched with a try construct. We omit the final ε in a sequence of definitions when it is not empty. The program is evaluated sequentially, until we arrive at the empty definition ε, where the program stops successfully, or until the program is the exceptional value $\mathsf{raise}\ v$, where the program stops with an error v.

Modules. We adopt the following simplified model of modules. A module μ is a collection containing two elements, a list of definitions *definitions*, and an interface which is a list of OCaml variables x defined in the definition list *definitions*. A module can depend on another modules, the free variables of the definition list must be included in the interfaces of the modules on which it depends.

Chapter 3

Translation

Our compiler automatically translates a specification written in the CryptoVerif language into OCaml. Let us describe this translation.

The annotations of Section 1.3 split the CryptoVerif code into multiple parts corresponding to different roles. Our compiler translates each of these roles role into an OCaml module μ_{role}. For each role role, let $Q(\mathsf{role})$ be the oracle declaration located between role [...] { and the following closing braces }. $Q(\mathsf{role})$ is the CryptoVerif code for the role role. Our compiler translates the oracles of $Q(\mathsf{role})$ into OCaml functions. More precisely, the implementation of the module μ_{role} consists of the init function, which reads the values of the variables required by the oracles in $Q(\mathsf{role})$ from the files, and returns the functions corresponding to the oracles declared by $Q(\mathsf{role})$. Functions corresponding to the oracles declared after a return in $Q(\mathsf{role})$ are not returned by init, but will be returned by that return, like continuations. Hence, the available functions correspond exactly to the oracles that can be called.

Example 7 Suppose that the role role is defined by

$$\mathsf{role}\ [\ldots]\ \{O_1(\ldots) := \ldots ; \mathsf{return}(M_1); O_2(\ldots) := \ldots ; \mathsf{return}(M_2)\}$$

Then the generated OCaml module μ_{role} provides a function init that returns a function that implements oracle O_1. When this function is called, it returns both the result of the oracle O_1 (the value of M_1) and the function that implements oracle O_2. That function just returns the result of O_2, that is, the value of M_2. □

This translation requires us to restrict the process when an oracle has several return statements: all these return statements must return data of the same type and oracles of the same name and type. We can work around this restriction as follows: when an oracle is missing at some return statements, we add a dummy oracle that ends immediately. As usual in functional languages, functions are represented by closures that contain a pointer to the code of the function and an environment that contains the free variables of the function. We rely on the OCaml type system to guarantee that the environment of closures is not accessed by the rest of the code, and in particular not sent directly to the adversary. The rest of this section details how the function init is generated.

For simplicity, we rename the variables in the CryptoVerif code in order to have a unique name for each variable. CryptoVerif already does this internally. Let $\mathbb{G}_{\mathsf{var}}$ be an injective function taking a CryptoVerif variable name, and returning an OCaml variable name. Let us also denote by T_M the type of a CryptoVerif term M.

The function $\mathbb{G}_{\mathsf{M}}$ transforms a term M into an OCaml term, in the obvious way:

$$\mathbb{G}_{\mathsf{M}}(x[\widetilde{i}]) \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{var}}(x)$$
$$\mathbb{G}_{\mathsf{M}}(f(M_1, \ldots, M_m)) \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{M}}(M_1))\ \ldots\ (\mathbb{G}_{\mathsf{M}}(M_m))$$

Lists are represented as follows. Let $[\,]$ be the empty list, and $x :: l$ be the list obtained by adding the element x to the list l. Let $[x_1; \ldots; x_k]$ be the list $x_1 :: \ldots :: x_k :: [\,]$. Let $[x \in l \mid \mathit{Prop}(x)]$ be the list containing all elements x of l that satisfy the property $\mathit{Prop}(x)$, in the same order as in l. This construct is defined by induction on lists:

$$[x \in [\,] \mid \mathit{Prop}(x)] \stackrel{\text{def}}{=} [\,],$$
$$[x \in y :: l \mid \mathit{Prop}(x)] \stackrel{\text{def}}{=} \begin{cases} [x \in l \mid \mathit{Prop}(x)] & \text{if } \neg \mathit{Prop}(y), \\ y :: [x \in l \mid \mathit{Prop}(x)] & \text{otherwise}. \end{cases}$$

The concatenation of lists $l_1 @ l_2$ is the list containing all elements of l_1 followed by all elements of l_2. The membership test $x \in l$ is true when l contains the element x, and false otherwise. Let $|l|$ be the length of the list l, and $\mathsf{nth}(l, n)$ be the nth element of list l.

The function reduce$'$ takes an oracle declaration Q and returns a list containing the oracles declared in Q without entering into oracle bodies. For each oracle, it also returns a boolean that is true when the oracle is defined under foreach (so can be called several times), and false otherwise. This function is defined as follows.

$$\text{reduce}'(0) \stackrel{\text{def}}{=} [\,]$$
$$\text{reduce}'(Q_1 \mid Q_2) \stackrel{\text{def}}{=} \text{reduce}'(Q_1) @ \text{reduce}'(Q_2)$$
$$\text{reduce}'(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} [(Q_1, \text{true}); \ldots; (Q_k, \text{true})] \text{ when}$$
$$[(Q_1, b_1); \ldots; (Q_k, b_k)] = \text{reduce}'(Q) \text{ for some } b_1, \ldots, b_k$$
$$\text{reduce}'(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\text{def}}{=}$$
$$[(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P, \text{false})]$$

This function is used in the generation of the init function in order to determine the oracles we can call at the beginning of the role, and in the translation of the return statement to determine which closures to give back to the caller.

In Figure 3.1, we define the function $\mathbb{G}$ that translates an oracle body into an OCaml term, as explained below.

As mentioned in Section 1.3, a role is declared with variables read from and written to files. Let write_file be an OCaml function of type string → string → unit that takes a file name and the contents to write and writes the contents to

Figure 3.1 Translation function $\mathbb{G}$ of an oracle body in OCaml

$$\mathbb{G}(x[\widetilde{i}] \xleftarrow{R} T; P) \stackrel{\text{def}}{=} \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{random}}(T)\ ()\ \mathsf{in}\ \mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]);\ \mathbb{G}(P) \tag{New}$$

$$\mathbb{G}(x[\widetilde{i}] \leftarrow M; P) \stackrel{\text{def}}{=} \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{in}\ \mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]);\ \mathbb{G}(P) \tag{Let}$$

$$\mathbb{G}(\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=} \mathsf{if}\ \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{then}\ \mathbb{G}(P)\ \mathsf{else}\ \mathbb{G}(P') \tag{If}$$

$$\mathbb{G}(\mathsf{event}\ e(M_1, \ldots, M_k); P) \stackrel{\text{def}}{=} \mathbb{G}(P) \tag{Event}$$

$$\begin{aligned}\mathbb{G}(\mathsf{return}(N_1, \ldots, N_k); Q) \stackrel{\text{def}}{=} (&\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_l, b_l),\\ &\mathbb{G}_{\mathsf{M}}(N_1), \ldots, \mathbb{G}_{\mathsf{M}}(N_k))\\ &\text{when reduce}'(Q) = (Q_1, b_1), \ldots, (Q_l, b_l)\end{aligned} \tag{Return}$$

$$\mathbb{G}(\mathsf{end}) \stackrel{\text{def}}{=} \mathsf{raise\ Match\_failure} \tag{End}$$

$$\frac{(Tbl, f) \in \mathsf{tables}}{\begin{array}{l}\mathbb{G}(\mathsf{insert}\ Tbl(M_1, \ldots, M_k); P) \stackrel{\text{def}}{=}\\ \quad \text{add\_to\_table}\ f\ (\mathbb{G}_{\mathsf{ser}}(T_{M_1})\ \mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{ser}}(T_{M_k})\ \mathbb{G}_{\mathsf{M}}(M_k));\\ \quad \mathbb{G}(P)\end{array}} \tag{Insert}$$

$$\begin{array}{l}\mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M) \stackrel{\text{def}}{=}\\ \quad (\mathsf{function}\ [\mathbb{G}_{\mathsf{var}}(x_1); \ldots; \mathbb{G}_{\mathsf{var}}(x_k)] \rightarrow\\ \qquad \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x_1) = \mathbb{G}_{\mathsf{deser}}(T_{x_1})\ \mathbb{G}_{\mathsf{var}}(x_1)\ \mathsf{in}\ \ldots\\ \qquad \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x_k) = \mathbb{G}_{\mathsf{deser}}(T_{x_k})\ \mathbb{G}_{\mathsf{var}}(x_k)\ \mathsf{in}\\ \qquad \mathsf{if}\ \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\\ \qquad \mathsf{else\ raise\ Match\_failure}\\ \quad \mid \_ \rightarrow \mathsf{raise\ Bad\_file})\end{array} \tag{Test}$$

$$\frac{(Tbl, f) \in \mathsf{tables}}{\begin{array}{l}\mathbb{G}(\mathsf{get}\ Tbl(x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=}\\ \quad \mathsf{let}\ list = \text{read\_table}\ f_{Tbl}\ \mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M)\ \mathsf{in}\\ \quad \mathsf{if}\ list = [\,]\ \mathsf{then}\ \mathbb{G}(P')\\ \quad \mathsf{else\ let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = \mathsf{random}_l\ list\ \mathsf{in}\\ \qquad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P))\end{array}} \tag{Get}$$

the file, and read_file a function of type string → string that takes a file name and returns its contents. We define a function $\mathbb{G}_{\mathsf{file}}$ that writes a variable to a file when needed: $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]) =$ write_file f $(\mathbb{G}_{\mathsf{ser}}(T_{x[\widetilde{i}]})\ \mathbb{G}_{\mathsf{var}}(x))$ when variable $x[\widetilde{i}]$ is written to file f in role role, that is, role is annotated with $x > f$, and $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]) = ()$ when x is not written to a file.

We translate $x[\widetilde{i}] \stackrel{R}{\leftarrow} T; P$ by binding the variable $\mathbb{G}_{\mathsf{var}}(x)$ to a random value in the type T, then writing its contents to the appropriate file if required, and finally continuing on the translation of the rest of the process P. We translate $x[\widetilde{i}] \leftarrow M; P$ in the same way, but we bind $\mathbb{G}_{\mathsf{var}}(x)$ to the result of $\mathbb{G}_{\mathsf{M}}(M)$, which is the translation of the CryptoVerif term M into OCaml. The translation of the if construct is straightforward. We simply ignore events in the translation, since they do not affect the execution of the system.

We translate the return statement into an OCaml tuple containing the closures of the oracles that become callable after that return (computed by the reduce′ function), and the translation of the terms $N_1, \ldots, N_k$. (The function $\mathbb{G}_{\mathsf{O}}$ is defined in Figure 3.2 and explained below.) end is translated into an exception because we need to stop the execution of the oracle here, and one must be able to distinguish whether we terminated on a return or on an end statement.

We translate the insert construct by simply adding to the appropriate file the serialization of the translation of arguments of insert. This translation uses the function add_to_table of type string → string list → unit, which takes a table file and a list of strings that represents an element of the table Tbl, and adds this element to the file. To translate a get construct, we use a function $\mathbb{G}_{\mathsf{collect}}((x_1, \ldots, x_k), M)$ that takes an element of the table, returns its deserialization if it satisfies M, and raises Match_failure otherwise. We also use a function read_table of type string → (string list → 'a) → 'a list such that read_table f_{Tbl} filter reads the table file f_{Tbl} and returns the list of values filter e for all elements e of the table such that filter e does not raise Match_failure. Therefore, by read_table f_{Tbl} $\mathbb{G}_{\mathsf{collect}}((x_1, \ldots, x_k), M)$, we collect all elements of the table that satisfy the term M. If there is no such element, we continue with the translation of the process P'. If there are such elements, we choose one of them randomly, we bind the variables $(\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))$ accordingly and add them to their respective files if necessary, and finally we continue with the translation of the process P.

An oracle $O(x_1, \ldots, x_n) := P$ is transformed into a closure by the function $\mathbb{G}_{\mathsf{O}}$ shown in Figure 3.2. The implementation differs depending on whether the oracle is under replication or not. If the oracle is not under replication, it must be callable at most once, so we create a new boolean reference that we store in *token*: *token* is true if and only if the oracle can still be executed. We initialize *token* to true. When we execute the oracle, we set *token* to false, to prevent other executions. The function also checks that its arguments are correct elements of their type by using the function $\mathbb{G}_{\mathsf{pred}}$, and then proceeds to execute the translation of the oracle body P. If the arguments are not correct elements of their type, or if the oracle is not under replication and has already been called, then it raises the exception Bad_Call without executing the translation of P.

The implementation of the module μ_{role} consists in the init function presented

Figure 3.2 Translation of an oracle

$$\begin{array}{l}
\mathbb{G}_{\mathsf{O}}(Q, \text{false}) \stackrel{\text{def}}{=} \mathsf{let}\ \mathit{token} = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{function}\ \mathit{pm}_{\text{false}}(Q) \\
\text{where } \mathit{pm}_{\text{false}}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\text{def}}{=} \\
\qquad (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) \rightarrow \\
\qquad \mathsf{if}\ (!\mathit{token})\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \ldots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k)) \\
\qquad \mathsf{then}\ (\mathit{token} := \text{false}; \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P)) \\
\qquad \mathsf{else\ raise\ Bad_Call}
\end{array} \tag{Oracle1}$$

$$\begin{array}{l}
\mathbb{G}_{\mathsf{O}}(Q, \text{true}) \stackrel{\text{def}}{=} \mathsf{function}\ \mathit{pm}_{\text{true}}(Q) \\
\text{where } \mathit{pm}_{\text{true}}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\text{def}}{=} \\
\qquad (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) \rightarrow \\
\qquad \mathsf{if}\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \ldots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k)) \\
\qquad \mathsf{then}\ (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P)) \\
\qquad \mathsf{else\ raise\ Bad_Call}
\end{array} \tag{Oracle2}$$

Figure 3.3 The init function for the module μ_{role}

Let $x_1 < f_1, \ldots, x_m < f_m$ be the annotations of role role that indicate variables read from files (explicit or implicit because of an annotation $x_i > f_i$ in a role above role when x_i is defined above role and used in role).
Let $\text{reduce}'(Q(\mathsf{role})) = (Q_1, b_1), \ldots, (Q_k, b_k)$.

$$\begin{array}{l}
\mathit{program}(\mu_{\mathsf{role}}) \stackrel{\text{def}}{=} \mathsf{let}\ \mu_{\mathsf{role}}.\mathsf{init} = \mathsf{let}\ \mathit{token} = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}}\ \mathit{pm}_{\mathsf{role}} \\
\text{where } \mathit{pm}_{\mathsf{role}} \stackrel{\text{def}}{=} () \rightarrow \\
\quad \mathsf{if}\ (!\mathit{token})\ \mathsf{then} \\
\qquad (\mathit{token} := \text{false}; \\
\qquad \mathbb{G}_{\mathsf{read}}(x_1[\,])\ \mathsf{in}\ \ldots\ \mathsf{in}\ \mathbb{G}_{\mathsf{read}}(x_m[\,])\ \mathsf{in} \\
\qquad (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_k, b_k))) \\
\quad \mathsf{else\ raise\ Bad_Call}
\end{array}$$

in Figure 3.3. It begins by reading all the required files, and then returns closures for all oracles that are callable at the beginning of the module. So, by calling this init function, the user gets access to the oracles present in the module. The init function can be called only once, as guaranteed by the boolean *token*.

Example 8 The role role_A whose process is:

$$\mathrm{OA}() := x \stackrel{R}{\leftarrow} nonce;\ s \stackrel{R}{\leftarrow} seed;\ \mathsf{return}(\mathrm{enc}(x, K_{ab}, s))$$

and reads the shared key K_{ab} from the file *keyfile* is translated in the OCaml module μ_{role_A} of Figure 3.4.

To use this module, the file *keyfile* must already have been generated. The network code calls the init function to get a closure of the oracle OA. Then it can call this closure with argument () to get back the encryption of the nonce x. The following OCaml code calls the init function and then calls the closure, and stores the encryption of x in r:

$$\mathsf{let}\ r = \mathsf{init}\ ()\ ()$$

The network code is then responsible for sending this encryption to B. □

Figure 3.4 The module μ_{role_A}

```
let init = let token = ref true in function () →
  if (!token) then
    (token := false;
     let G_var(K_ab) = G_deser(key) (read_file "keyfile") in
     let token = ref true in function () →
       if (!token) then
         (token := false;
          let G_var(x) = G_random(nonce) () in
          let G_var(s) = G_random(seed) in
          (G_f(enc) (G_var(x), G_var(K_ab), G_var(s))))
       else raise Bad_Call)
  else raise Bad_Call
```

(OA)

To make sure that this implementation behaves as expected, the network code, which is manually written and calls this implementation, must satisfy certain constraints. This code must not use unsafe OCaml functions (such as Obj.magic or marshalling/unmarshalling with different types) to bypass the typesystem (in particular to access the environment of closures). We also require that this code does not mutate the values received from or passed to functions generated by CryptoVerif. This can be guaranteed by using immutable types, with the above requirement that the typesystem is not bypassed. However, OCaml typically uses string for cryptographic functions and for network input/output, and the type string is mutable in OCaml. For simplicity and efficiency, the generated code uses the type string, with the no-mutation requirement above. We also require that

all data structures manipulated by the generated code are non-circular. This is necessary because we use the OCaml structural equality to compare values, and this equality may not terminate in the presence of circular data structures. This can be guaranteed by requiring that all OCaml types corresponding to CryptoVerif types are non-recursive. We also require that the network code does not fork after obtaining but before calling an oracle that can be called only once (because it is not under a replication in the CryptoVerif specification). Indeed, forking at this point would allow the oracle to be called several times. In general, forking occurs only at the very beginning of the protocol, when the server starts a new session, so this requirement should be easily fulfilled. These requirements could be verified by program analysis.

Finally, we require that the roles are executed in the order specified by the CryptoVerif specification. For instance, in general, the key generation programs must be executed before the client and the server.

Example 9 One can design a protocol where the adversary would gain information if he could initialize roles he should not have access to. Consider the following protocol.

$$\begin{array}{l}\mathsf{role}[x > f]\,\{O() := \\ \qquad\qquad secret \xleftarrow{R} bool; public \xleftarrow{R} bool; \\ \qquad\qquad \mathsf{if}\ secret \,||\, public\ \mathsf{then} \\ \qquad\qquad\quad \mathsf{let}\ x = \mathrm{true}\ \mathsf{in} \\ \qquad\qquad\quad \mathsf{if}\ public\ \mathsf{then}\ \mathsf{return}()\}; \\ \qquad \mathsf{test}\,\{O'() := \mathsf{return}(x)\end{array}$$

The test role can normally only be executed when the oracle O finishes on its return() statement, which is the case when the variable *public* is true. As an adversary can distinguish between whether an oracle exits on a return or an end statement (which is present in O in the implicit else branch of the if statement else end), the value of *public* is known as soon as the role O finishes.

If an adversary executes our module μ_{test} that implements role test when the oracle O finishes on the end statement, either the init function of the module μ_{test} initializes successfully, which would mean that x was defined and that *secret* is true or it would raise an error in function read_file because the file f containing variable x does not exist. So the adversary is able to discover the value of the secret boolean *secret*. □

This example has been crafted to show the importance of this assumption. But, roles in most protocols can be correctly initialized only when they can be run, e.g., in a key exchange protocol, the client and the server can only be run when the key generation has correctly returned the keys. Also, different roles are designed to run on different machines, so it is not practical to add a means of synchronization (for example using files, like what we did for passing values from a role to an other) to know which oracles are callable.

We also require that several programs that insert elements in the same table are not run concurrently, to avoid conflicting writes. This requirement could be enforced using locks, but in practice, it is generally obtained for free if the programs are run in the intended order.

We also require that the files used by the generated code are not read or written by other software, as this could obviously break security.

We will formalize these assumptions in Chapter 7.

Chapter 4

SSH Transport Layer Protocol

This chapter applies our work to an implementation of the Secure Shell (SSH) protocol. We first present the protocol, then present our model, the proofs of the security properties, and the generated implementation.

4.1 Description of the SSH Protocol

The SSH protocol is a protocol that permits a client to contact a server and run an application on it securely. When a session is established, the client and the server are authenticated and data runs through a secure channel to ensure its privacy and integrity.

SSH (version 2.0) is divided in three parts [50]. The SSH Transport Layer Protocol [52] authenticates the server to the client and establishes a secure tunnel for the other parts. This secure tunnel is implemented using encryption and MAC (message authentication code), with keys chosen by a Diffie-Hellman key exchange. The tunnel aims to guarantee the privacy and integrity of the data going through. The SSH Authentication Protocol [51] authenticates the client. The SSH Connection Protocol [53] multiplexes multiple channels through the tunnel.

We concentrated our efforts on the Transport Layer part. In Figure 4.1, we present an overview of this part. The key exchange part consists of four groups of messages:

1. The client and the server send their identification string, which specifies the version of SSH they use.

2. Then the server sends to the client the lists of the cryptographic algorithms for key exchange, signature, encryption, MAC, and compression it can use in order of preference, and the client sends the list of cryptographic algorithms it supports. Based on this information, the protocol chooses which algorithms to use. Our implementation uses diffie-hellman-group14-sha1, RSA signature, AES128-CBC, HMAC-SHA1, and no compression as algorithms, respectively. SSH specifies other algorithms as well. Most of them would be very easy to include in our implementation; still, the additional counter modes encryptions specified in [10] raise an additional difficulty as discussed below in Section 4.5.

Figure 4.1 Overview of the SSH Transport Layer Protocol

Key exchange:

	Client C	Message	Server S
1.		$\xrightarrow{id_C=\text{SSH-2.0-version}_C}$	
		$\xleftarrow{id_S=\text{SSH-2.0-version}_S}$	
2.		$\xrightarrow{\text{KEX\_INIT},cookie_C,algos_C}$	
		$\xleftarrow{\text{KEX\_INIT},cookie_S,algos_S}$	
3.	$x \xleftarrow{R} [2, q-1], e = g^x$	$\xrightarrow{\text{KEXDH\_INIT},e}$	$y \xleftarrow{R} [1, q-1], f = g^y$
	$K = f^x$	$\xleftarrow{\text{KEXDH\_REPLY},pk_S,f,\text{sign}(H,sk_S)}$	$K = e^y$
4.	$pk_S, \text{sign}(H, sk_S)$ ok?	$\xrightarrow{\text{NEWKEYS}}$	
		$\xleftarrow{\text{NEWKEYS}}$	

where $H = \text{SHA1}(id_C, id_S, cookie_C, algos_C, cookie_S, algos_S, pk_S, e, f, K)$

Tunnel keys:

$$\begin{aligned} sessionid &= H \\ IV_C &= \text{SHA1}(K, H, \text{"A"}, sessionid) \\ IV_S &= \text{SHA1}(K, H, \text{"B"}, sessionid) \\ K_{\text{enc},C} &= \text{SHA1}(K, H, \text{"C"}, sessionid) \\ K_{\text{enc},S} &= \text{SHA1}(K, H, \text{"D"}, sessionid) \\ K_{\text{MAC},C} &= \text{SHA1}(K, H, \text{"E"}, sessionid) \\ K_{\text{MAC},S} &= \text{SHA1}(K, H, \text{"F"}, sessionid) \end{aligned}$$

Tunnel:

$$\text{Client } C \quad \begin{array}{l} \xrightarrow{\text{enc}(K_{\text{enc},C},packet,IV_C),\text{MAC}(K_{\text{MAC},C},sequence\_number_C \| packet)} \\ \xleftarrow{\text{enc}(K_{\text{enc},S},packet,IV_S),\text{MAC}(K_{\text{MAC},S},sequence\_number_S \| packet)} \end{array} \quad \text{Server } S$$

where $packet = packet\_length \| padding\_length \| payload \| padding$

3. Then the actual key exchange takes place. The key exchange messages depend on the chosen key exchange algorithm. The algorithm we use relies on a group defined in [34]. Let p be a large prime and g be a generator of a subgroup of $\mathbb{Z}_p^\star$.

   First, the client chooses a random exponent x and sends to the server $e = g^x \bmod p$.

   Then the server chooses a random exponent y and computes $f = g^y \bmod p$, the shared key $K = e^y \bmod p$, and the SHA1 hash H of the messages previously sent by the client and the server, the server public host key pk_s, f, and K. It then signs this hash with its private host key sk_S. Let $s = \mathrm{sign}(H, sk_S)$ be this signature. It finally sends back pk_s, f, and s.

   The client must then verify that pk_s is indeed the key for the server it intended to reach, then compute the shared key $K = f^x \bmod p$, the hash H in the same manner as the server, and then verify the signature.

4. When the client has verified the server's message, it sends a "new keys" message declaring that the key they agreed upon is to be used afterwards, and the server acknowledges this by also sending the same message.

   From the values of H and K, SSH then generates two encryption keys (one for client to server messages, and one for server to client messages) $K_{\mathrm{enc},C}$ and $K_{\mathrm{enc},S}$, two initialization vectors (IVs) for the encryption IV_C and IV_S, and two keys for MAC $K_{\mathrm{MAC},C}$ and $K_{\mathrm{MAC},S}$, by computing hashes of H, K, and different constants. The forthcoming messages in the SSH protocol will be encrypted and a MAC will be computed based on the clear message and on a sequence number that is incremented at each message.

Each message of the protocol, except the identification string messages, begins with five bytes indicating the size of the message (first four bytes) and the size of the random padding (one byte) present after the message, and is padded to a multiple of the block size of the encryption scheme (or 8, at the beginning when the encryption scheme is not chosen yet).

4.2 Our Model of SSH in CryptoVerif

We have modeled the SSH Transport Layer Protocol in the CryptoVerif specification language. In our model, the first role corresponds to the key generation. Its oracle generates the public/private key pair pk_S, sk_S of the server, and returns the public part of the key to the adversary. After the execution of this role, one can execute N times a client and N times a server.

To illustrate our model of SSH, we give the server process in Figures 4.2 and 4.3. An adversary can give to the protocol malformed messages, so that the server and the client may have different values for the same variable. So, for a given variable x of the protocol, we denote by x_S the variable the server uses to hold x, and by x_C the variable the client uses to hold x.

The protocol begins by exchanging the identification strings. Since this exchange requires no cryptography, it is not included in the CryptoVerif model

Figure 4.2 The server role in the SSH model (first part)

```
1  negotiation_S() :=
2      cookie_SS <-R- cookie;
3      init_SS <- concatm(KEX_INIT,
                          concat_KEX_INIT(cookie_SS, negotiation_string));
4      return(pad(init_SS));
5  key_exchange_2S(id_CS : bitstring, id_SS : bitstring,
                   m_1 : bitstring, m_2 : bitstring) :=
6      let injbot(init_CS) = unpad(m_1) in
7      let concatm(=KEX_INIT,
                   concat_KEX_INIT(cookie_CS, ns_CS)) = init_CS in
8      if (check_algorithms(ns_CS)) then
9      let injbot(concatm(=KEXDH_INIT, bitstring_of_G(e_S))) =
         unpad(m_2) in
10     y_S <-R- Z;  f_S <- exp(g, y_S);  K_S <- exp(e_S, y_S);
11     H_S <- hash(hk, concat8(id_CS, id_SS, init_CS, init_SS, pk_S, e_S, f_S, K_S));
12     event endS(id_CS, id_SS, init_CS, init_SS, pk_S, e_S, f_S, K_S, H_S);
13     s_S <- sign(block_of_hash(H_S), sk_S);
14     return(pad(concatm(KEXDH_REPLY,
                          concat_KEXDH_REPLY(pk_S, f_S, s_S))));
15 key_exchange_4S(m : bitstring) :=
16     let injbot(nk_CS) = unpad(m) in
17     let concatm(=NEWKEYS, =null_string) = nk_CS in
18     return(pad(concatm(NEWKEYS, null_string)));
19 get_keys_S() :=
20     IV_CS <- genIV_C(hk, K_S, H_S, H_S);
21     IV_SS <- genIV_S(hk, K_S, H_S, H_S);
22     K_enc,CS <- genK_enc,C(hk, K_S, H_S, H_S);
23     K_enc,SS <- genK_enc,S(hk, K_S, H_S, H_S);
24     K_MAC,CS <- genK_MAC,C(hk, K_S, H_S, H_S);
25     K_MAC,SS <- genK_MAC,S(hk, K_S, H_S, H_S);
26     return(IV_CS, IV_SS, H_S);
```

Figure 4.3 The server role in the SSH model (second part)

```
27 ( foreach j ≤ N' do
28     tunnel_send_S(payload : bitstring, IV_S : IV,
                     sequence_number_S : uint32) :=
29         packet ← pad(payload);
30         return(concatem(enc(packet, K_enc,SS, IV_S),
               mac(concatnm(sequence_number_S, packet), K_MAC,SS)))
31 | foreach j ≤ N' do
32     tunnel_recv_1S(m : bitstring, IV_C : IV) :=
33         let injbot(m_1) = dec(m, K_enc,CS, IV_C) in
34         return(get_size(m_1));
35     tunnel_recv_2S(m : bitstring, IV_C : IV, m' : mac,
                      sequence_number_C : uint32) :=
36         let injbot(m_2) = dec(m, K_enc,CS, IV_C) in
37         let packet = concat(m_1, m_2) in
38         if (check_mac(concatnm(sequence_number_C, packet),
                         K_MAC,CS, m')) then
39         let injbot(payload) = unpad(packet) in
40         return(payload)).
```

but is done by the network code, part of the adversary. The identification strings id_C and id_S are given as argument to the first oracle that requires them; hence, on Line 5, the oracle key_exchange$_{2S}$ takes id_{CS} and id_{SS} as arguments.

Then in the protocol, we have the algorithms negotiation phase, that is done in the first oracle negotiations$_S$ on Line 1. It first generates a random cookie $cookie_{SS}$. The function concatm is an injective function that concatenates a message tag with a bitstring. All functions whose name begins with concat are injective functions that concatenate their arguments. On Line 3, we create the payload of the negotiation packet using these concatenation functions. Then, we pad the payload accordingly to the specification with function pad to get a packet that we return on Line 4 to the adversary. This part cannot be done by the adversary, because we need to be sure that the cookie is randomly generated.

Then, the client sends in a similar KEX_INIT packet containing the algorithms the client supports and the cookie of the client. It then sends the first message of the key exchange KEXDH_INIT. The server must then send back the next message KEXDH_REPLY. This is done in the oracle key_exchange$_{2S}$ on Line 5. It takes id_{CS}, id_{SS} as we said before, and the packets m_1 and m_2 corresponding to the KEX_INIT and KEXDH_INIT messages of the client. We first obtain the payload corresponding to the negotiation packet m_1 on Line 6 by using the function unpad. This function takes a packet and returns its payload if it is a valid packet and $\perp$ otherwise. Next, we verify on Line 7 that this payload is indeed a KEX_INIT message and we obtain the values of the client cookie $cookie_{CS}$ and the list of algorithms of the client ns_{CS}. Next, we verify that the algorithms are compatible with the algorithms of the server on

Line 8. We deconstruct the KEXDH_INIT packet m_2 as above, we randomly generate y_S, and compute f_S, K_S and H_S. The hash function hash takes a key hk that represents the choice of the algorithm of the hash function. (This key is present in the cryptographic model, but not in the implementation.) On Line 12, we execute the event endS that is used in the proof of authentication (see Section 4.3). We sign the hash H_S on Line 13, and return the KEXDH_REPLY packet on Line 14.

After verifying that this message is correct, the client sends back to the server a NEWKEYS packet. The oracle key_exchange$_{4S}$ on Line 15 takes this packet and also returns a NEWKEYS packet.

At this point, the client and the server have agreed upon the values K and H to create the tunnel IVs, encryption and MAC keys. The oracle get_keys$_S$ on Line 19 takes nothing, and computes these keys. The function genIV_C is defined as follows:

$$\mathsf{letfun}\ \mathrm{gen}IV_C(hk : hkey, K : G, H : hash, sid : hash) = \\ \mathrm{iv\_of\_hash}(\mathrm{hash}(hk, \mathrm{concat4}(K, H, \texttt{"}A\texttt{"}, sid))).$$

The function genIV_C generates the SHA1 hash as shown in Figure 4.1 (Tunnel keys), and truncates this hash by function iv_of_hash to obtain a valid IV. The other IV, and the encryption and MAC keys are computed in a similar way. Next, we return the IVs and the session identifier to the network code on Line 26. SSH with AES128-CBC (or other CBC mode encryptions) uses CBC mode [36, Section 7.2.2 (ii)] with chained IVs, that is, the IV for the next message is the last block of ciphertext. Since CryptoVerif does not allow maintaining a mutable state across several oracle invocations, we simply get the IV from the network code which keeps in memory the last block of ciphertext it saw. That is why we return the initial IVs to the network code. We also return the session identifier, which is required in the next parts of the protocol that we implemented in the network code.

We model the SSH tunnel by oracles that get an encrypted packet from the network and return the clear payload to the application, and get a clear payload from the application and return the corresponding encrypted packet to the network code. After the return on Line 26, we can call the tunnel sending and receiving parts N' times.

Sending a packet is implemented by the oracle tunnel_send$_S$ on Line 28 taking a payload, the current server to client IV, and the sequence number. We need to pass the sequence number as argument, since we cannot keep it in a state in CryptoVerif. We pad the payload, yielding a packet. We encrypt this packet, append the MAC of the sequence number and the packet, and return the obtained message on Line 30.

The packets after the key exchange are completely encrypted under the key derived from the key exchange, the first five bytes containing the size of the packet included. Therefore, an implementation must decrypt the first block of the packet to get its size, then input the rest of the packet, decrypt it, and then check that the MAC that follows in the stream is correct. So we implemented receiving a packet by two successive oracles: first, the oracle tunnel_recv$_{1S}$ on Line 32 that takes the first block of the packet and the current client to server

IV, decrypts this block, and returns the size of the packet on Line 34. The network code can then input a packet of the required length, and call the second oracle tunnel_recv$_{2S}$ on Line 35 that takes the rest of the packet, its MAC, IV, and the sequence number corresponding to this message, checks the MAC and returns the decrypted payload if the MAC is correct.

4.3 Proof of Authentication of the Server

We have proved the authentication of the server in the computational model automatically by using CryptoVerif, assuming the RSA signature is UF-CMA (unforgeable under chosen message attacks) and the SHA1 hash function is collision-resistant. The authentication property shows that each session of the client C with the server S corresponds to a distinct session of the server S with the client C, and that the client C and the server S share all protocol parameters: identification strings, algorithm lists, pk_S, e, f, K, and H.

More formally, we define the events:

event $endC(bitstring, bitstring, bitstring, bitstring, spkey, G, G, G, hash)$.
event $endS(bitstring, bitstring, bitstring, bitstring, spkey, G, G, G, hash)$.

where event *endC* occurs in the client just after he verifies the signature of the server, and event *endS* occurs in the server just after he computes H_S (line 12 of Figure 4.2). The first four arguments of these events correspond to the messages exchanged in the session, two for the identification strings and two for the negotiation messages. The fifth argument corresponds to the public key of the server. The sixth and seventh arguments are the group elements e and f. So the first seven messages correspond to the messages passed between the client and the server in a session until the end of the key exchange phase. The eighth argument corresponds to the shared key K and the last argument corresponds to the hash H.

We ask CryptoVerif to prove the following properties:

$$\begin{array}{l}\forall vc : bitstring, vs : bitstring, ic : bitstring, is : bitstring, pk : spkey, x : G,\\ y : G, k : G, h : hash;\\ \quad \mathsf{inj}\text{: } endC(vc, vs, ic, is, pk, x, y, k, h) \Rightarrow \mathsf{inj}\text{: } endS(vc, vs, ic, is, pk, x, y, k, h)\,,\end{array} \tag{4.1}$$

$$\begin{array}{l}\forall vc : bitstring, vs : bitstring, ic : bitstring, is : bitstring, pk : spkey, x : G,\\ y : G, k : G, h : hash, k' : G, h' : hash;\\ \quad endC(vc, vs, ic, is, pk, x, y, k, h) \wedge endS(vc, vs, ic, is, pk, x, y, k', h') \Rightarrow\\ \qquad k = k' \wedge h = h'\,.\end{array} \tag{4.2}$$

Property (4.1) means that each execution of event *endC* corresponds to a distinct execution of event *endS*, with the same arguments. (The indication inj: means that the correspondence is injective, that is, two executions of *endC* cannot correspond to the same execution of *endS*.) Property (4.2) means that, if events *endC* and *endS* are executed with the same first seven arguments, then their

last two arguments are also the same, that is, if the client and server exchange the same public messages, then the key and the hash they compute are the same.

The proof found by CryptoVerif is the following:

1. CryptoVerif first simplifies the initial game. In particular, it transforms the insert and get constructs into find constructs.

2. After these transformations, it can prove Property (4.2), because, if the first seven arguments of the events *endC* and *endS* are equal, the eighth and ninth are computed in the same manner from the first seven, so they are equal.

3. Next, CryptoVerif replaces the secret and public keys of the server, sk_S and pk_S, with their values, sskgen(r) and spkgen(r) respectively, where r is a random number and sskgen and spkgen are the key generation functions for the signature scheme. This replacement allows CryptoVerif to apply the security assumption on the signature in the next step.

4. Next, CryptoVerif transforms the game by relying on the assumption that the signature scheme is UF-CMA. Indeed, by the UF-CMA property, up to negligible probability, the adversary cannot forge a signature, so the verification of the signature in the client, check(m_C, pk_{SC}, s_C) where m_C = block_of_hash(H_C) and $pk_{SC} = pk_S$, can succeed only if the message m_C has been signed under sk_S. Moreover, the only signature under sk_S occurs in the server (line 13 of Figure 4.2). CryptoVerif transforms this signature by first storing block_of_hash(H_S) in m_S, then computing sign(m_S, sk_S). It replaces the verification of the signature in the client, check(m_C, pk_{SC}, s_C), with a find that looks for a signature of m_C under sk_S, that is, a find that looks for a session u of the server such that $m_S[u]$ is defined and $m_S[u] = m_C$. (Recall that variables are implicitly arrays; $m_S[u]$ is the value of m_S in session u of the server.)

5. The obtained game is then simplified. In particular, the equality $m_S[u] = m_C$ above becomes block_of_hash($H_S[u]$) = block_of_hash(H_C), that is, $H_S[u] = H_C$ since block_of_hash is injective. Hence, this equality becomes hash(hk, concat8($id_{CS}[u], id_{SS}[u], init_{CS}[u], init_{SS}[u], pk_S, e_S[u], f_S[u], K_S[u]$)) = hash($hk$, concat8($id_{CC}, id_{SC}, init_{CC}, init_{SC}, pk_{SC}, e_C, f_C, K_C$)). Since hash is collision-resistant and concat8 is injective, this equality becomes $id_{CS}[u] = id_{CC} \wedge id_{SS}[u] = id_{SC} \wedge init_{CS}[u] = init_{CC} \wedge init_{SS}[u] = init_{SC} \wedge pk_S = pk_{SC} \wedge e_S[u] = e_C \wedge f_S[u] = f_C \wedge K_S[u] = K_C$.

   CryptoVerif can then prove Property (4.1). In the initial game, the event *endC* is located after the signature verification in the client. Therefore, when the client executes event $endC(id_{CC}, id_{SC}, init_{CC}, init_{SC}, pk_{SC}, e_C, f_C, K_C, H_C)$, the find that replaces signature verification succeeds, so $m_S[u]$ is defined, which implies that the server has executed event $endS(id_{CS}[u], id_{SS}[u], init_{CS}[u], init_{SS}[u], pk_S, e_S[u], f_S[u], K_S[u], H_S[u])$ located above the definition of m_S, and the condition $m_S[u] = m_C$ simplified above holds, so the arguments of these events are equal. Moreover, two distinct

executions of *endC* have distinct arguments e_C up to negligible probability (because $e_C = g^x$ for a random exponent x), so they correspond to two distinct executions of *endS*, which proves injectivity.

4.4 Proof of Secrecy of the Session Keys

We have also proved the secrecy of the session keys obtained by key exchange (the encryption keys, MAC keys, and initialization vectors for encryption), that is, an adversary has a negligible probability of distinguishing these keys from random numbers, assuming the group used by the key exchange satisfies the CDH (Computational Diffie-Hellman) assumption, the SHA1 hash function is a random oracle, and the RSA signature is UF-CMA. This proof is performed on a protocol that stops just after key exchange, because the cryptographic secrecy of the keys is broken as soon as they are used by the protocol. Moreover, we prove secrecy for the keys computed by the client; the keys of the server are not always secret, because the server may also execute sessions with the adversary. The proof is performed by CryptoVerif with manual guidance of the user.

In our proof of secrecy of the session keys, we also prove the authentication property again assuming SHA1 is a random oracle (which implies collision resistance). With the random oracle model, we need to provide the adversary with a hash oracle O_H, so that it can compute hashes. This oracle O_H takes as argument a bitstring h and returns its hash:

$$O_H(h : bitstring) := \mathsf{return}(\mathrm{hash}(hk, h)) .$$

We provided CryptoVerif with proof indications to help the tool prove this property, as follows:

1. CryptoVerif first simplifies the initial game automatically. In particular, it transforms the insert and get constructs into find constructs.

2. By the command success, we ask CryptoVerif to try to prove the desired security properties. It manages to prove Property (4.2), as in Section 4.3. The other properties cannot be proved yet.

3. The hash function hash is used with two kinds of arguments. It is used to compute the hash H with the concatenation by concat8 of eight arguments corresponding to the messages in the current session, and it is also used to compute the generated keys with argument concat4(K, H, c, H) for several constants c. To simplify the game obtained after applying the random oracle assumption (Step 4), we distinguish in the hash oracle O_H these two uses of the hash function.

   By command

   $$\mathsf{insert}\ 350\ \mathsf{let}\ \mathrm{concat8}(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8) = h\ \mathsf{in}$$

   we add a let at the beginning of the oracle O_H. (The occurrence 350 corresponds to the beginning of O_H. Occurrence numbers for each program

point in the game can be shown by command show_game occ.) Then O_H becomes let concat8$(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8) = h$ in return(hash(hk, h)) else return(hash(hk, h)). We also insert let concat4$(b_1, b_2, b_3, b_4) = h$ in in the else branch of the previous let. So the oracle is now split in three, where the first part is for concat8 arguments, the second part for concat4 arguments, and the last part for other arguments.

4. By command crypto rom(hash), we apply the random oracle assumption on the function hash. CryptoVerif transforms each call to the function hash into a lookup in the previous calls to the function hash: if the same hash has already been computed, we return the same result; otherwise, we return a fresh random hash.

5. By command crypto uf_cma(signr), we apply the UF-CMA transformation after replacing sk_S and pk_S with their values, as in Section 4.3, and we simplify the obtained game. (The function signr is the deterministic signature function, which takes random coins as argument. The function sign is defined by letfun as in Example 6. It generates random coins and calls signr with these coins.)

6. By command success, CryptoVerif proves Property (4.1). Instead of using collision resistance as it did in Section 4.3, it relies on the negligible probability of collisions between fresh random hashes.

7. In order to prove the secrecy of the keys generated by the client, we want to transform the game using the CDH assumption. This transformation basically transforms equality tests of the form $M = \exp(g, \text{mult}(x, y))$ into false when the only usages of the random exponents x and y are for computing $\exp(g, x)$, $\exp(g, y)$ and equality tests of the form $M = \exp(g, \text{mult}(x, y))$. Indeed, the CDH assumption says that one has a negligible probability of computing M such that $M = \exp(g, \text{mult}(x, y))$ knowing $\exp(g, x)$, $\exp(g, y)$ for random exponents x and y. In order to use this transformation, we need to eliminate as many usages of x and y as possible.

   The game contains a process of the following form in the client:

$$\begin{array}{l} K_C \leftarrow \exp(f_C, x_C); \\ \mathsf{find}\ w' \leq N\ \mathsf{suchthat}\ \mathsf{defined}(id_{CS}[w'], \ldots, f_S[w']) \wedge \\ \quad id_{CC} = id_{CS}[w'] \wedge \ldots \wedge f_C = f_S[w']\ \mathsf{then} \\ \quad \ldots IV_{CC} \leftarrow \ldots \\ \oplus\ w'' \leq N_H\ \mathsf{suchthat}\ \mathsf{defined}(a_1[w''], \ldots, a_8[w'']) \wedge \\ \quad id_{CC} = a_1[w''] \wedge \ldots \wedge K_C = a_8[w'']\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots \end{array} \tag{4.3}$$

   This find tests whether there exists a session of the server indexed by w' that has exactly the same messages as the current session of the client, and if this is the case, it generates the session keys. In the other then branch and in the else branch of this find, we do not generate the keys.

We use an insert command to add the following find:

$$\begin{array}{c}\mathsf{find}\ w \leq N\ \mathsf{suchthat}\ \mathsf{defined}(id_{CS}[w], \ldots, f_S[w]) \wedge \\ id_{CC} = id_{CS}[w] \wedge \ldots \wedge f_C = f_S[w]\ \mathsf{then}\end{array} \tag{4.4}$$

above the definition of K_C in (4.3). The rest of the code following the find (4.4) is duplicated in the then and else branches of that find.

In subsequent simplifications (Step 9 below), in the find (4.3) that occurs in the else branch of (4.4), the first then branch is removed, because when we take the else branch of (4.4), no w satisfying the condition can be found, so also no w'' satisfying the same condition, hence we never take the first then branch of (4.3). The find (4.3) that occurs in the then branch of (4.4) is transformed into its first then branch, because when we take the then branch of (4.4), (4.3) always finds a w' equal to w. As a result, the usage of K_C in $K_C = a_8[w'']$ disappears when the condition of (4.4) holds. All other usages of K_C when this condition holds are of the form $M = \exp(g, \mathrm{mult}(x, y))$, so they can be handled by the CDH assumption.

8. To continue helping CryptoVerif remove usages of x and y, we distinguish cases depending on whether the server runs a session with the honest client or with the adversary. We use an insert command to insert the find:

$$\mathsf{find}\ v \leq N\ \mathsf{suchthat}\ \mathsf{defined}(e_C[v]) \wedge e_C[v] = e_S\ \mathsf{then} \tag{4.5}$$

before the creation of the shared Diffie-Hellman key K_S in the server (middle of line 10 in Figure 4.2). This allows us to distinguish the case in which the group element e_S of the server comes from the client (then branch) from the case in which e_S comes from the adversary (else branch).

9. By command simplify, CryptoVerif simplifies the game. In particular, it renames the variable K_S into two variables, K_{S1} for the variable K_S defined in the then branch of the find (4.5) and K_{S2} for the one defined in the else branch of this find.

10. By command crypto cdh(exp), we transform the game using the CDH assumption. CryptoVerif automatically performs some preparatory steps before actually using CDH, and simplifies the game after applying CDH.

After applying CDH, we arrive at a game of the following form:

$\mathsf{foreach}\ i \leq N\ \mathsf{do}\ \ldots\ \mathrm{key\_exchange}_{2S}(\ldots) := \ldots$
$\quad \mathsf{find}\ v \leq N\ \mathsf{suchthat}\ \mathsf{defined}(e_C[v]) \wedge e_C[v] = e_S\ \mathsf{then}\ \ldots$
$\quad \mathsf{else}\ K_{S2} \leftarrow \exp(e_S, y_S); \ldots$

(comes from (4.5))

$\mid \mathsf{foreach}\ j \leq N\ \mathsf{do}\ \ldots\ \mathrm{key\_exchange}_{1C}(\ldots) := \ldots$
$\quad x_C \stackrel{R}{\leftarrow} Z; e_C \leftarrow \exp(g, x_C); \ldots$
$\quad \mathsf{find}\ w \leq n\ \mathsf{suchthat}\ \mathsf{defined}(e_S[w], \ldots) \wedge \ldots \wedge (e_C = e_S[w])\ \mathsf{then}$
$\quad\quad \ldots$
$\quad\quad \mathsf{if}\ \mathsf{defined}(K_{S2}[w])\ \mathsf{then}\ \ldots\ \mathsf{else}\ P$

(comes from (4.4))

Here we use our extension presented in Section 1.5.2; we prove that the condition $\mathsf{defined}(K_{S2}[w])$ of the last test cannot be satisfied.

Assuming that we reach the last test and $K_{S2}[w]$ is defined, we have taken the else branch of the find in key_exchange$_{2S}$ in the run of index w, so at the definition of $K_{S2}[w]$, we have that, for all v, $\mathsf{defined}(e_C[v]) \wedge e_C[v] = e_S[w]$ does not hold. We have two cases:

- If the variable $e_C[j]$ (the value of e_C with the current index j, also denoted e_C) is defined before $K_{S2}[w]$, then at the definition of $K_{S2}[w]$, $e_C[j]$ was defined and for all v, $\mathsf{defined}(e_C[v]) \wedge e_C[v] = e_S[w]$ does not hold. Taking $v = j$, $\mathsf{defined}(e_C[j]) \wedge e_C[j] = e_S[w]$ does not hold, so $e_C[j] \neq e_S[w]$: the condition of the last find construct is false in this case, so we cannot reach the last test.
- Otherwise, the variable $e_C[j]$ is defined after $K_{S2}[w]$, so $x_C[j]$ is defined after $e_S[w]$. Since $x_C[j]$ is chosen randomly after $e_S[w]$, it is independent of $e_S[w]$, so $e_C[j]$ is a random element chosen uniformly in G independent of $e_S[w]$. Therefore, the probability that $e_C[j] = e_S[w]$ is $1/|G|$. We eliminate this collision, which happens with negligible probability, so that we also have $e_C[j] \neq e_S[w]$, so we also cannot reach the last test.

This is a contradiction, so we cannot reach the last test with $K_{S2}[w]$ defined, hence we can replace this test with its else branch P, taking into account the collision probability $1/|G|$ in the probability of success of an attack.

In other words, when the client makes a successful run with the server (the signature verification succeeds, so the find (4.4) succeeds), then the server has also used an element e_S coming from the client, so we have taken the then branch of the find (4.5), so K_{S2} is not defined.

Basically, the transformations we did previously allow us to distinguish cases depending on whether the exponents x and y are used in sessions between the honest client and server, or they are used in sessions with the adversary. Furthermore, for exponents x and y used in sessions between the honest client and server, the only usages of x and y left after the previous transformations are of the form $\exp(g, x)$, $\exp(g, y)$, and $M = \exp(g, \mathrm{mult}(x, y))$. By the CDH assumption, these equality tests can then be replaced with false.

In particular, in the initial game, IVs (and session keys) are computed by formulas such as

$$IV_{CC} \leftarrow \mathrm{iv\_of\_hash}(\mathrm{hash}(hk, \mathrm{concat4}(K_C, H_C, \text{"A"}, H_C))) .$$

By the random oracle model, the call to hash is replaced with a find that compares $\mathrm{concat4}(K_C, H_C, \text{"A"}, H_C)$ to the arguments of the previous hash queries (Step 4). Due to the previous simplifications, it just compares $\mathrm{concat4}(K_C, H_C, \text{"A"}, H_C)$ to the hash queries $\mathrm{concat4}(b_1, b_2, b_3, b_4)$ made by the adversary in oracle O_H. In case the same arguments are found, we

compute the IV by truncating the result returned by the previous call to O_H, so in this case, the adversary would have the IV. Otherwise, we generate a fresh IV by returning iv_of_hash(r) for a random r. Importantly, the former case is removed by the CDH assumption, since the comparison $b_1 = K_C$ is of the form $M = \exp(g, \text{mult}(x, y))$, so it is false. Hence, in fact, we always generate a fresh IV by returning iv_of_hash(r) for a random r.

11. The function iv_of_hash truncates its input to the size of its output. Hence, if the argument of iv_of_hash is uniformly distributed, then so is its result. We give that information to CryptoVerif by adding the transformation hash_to_iv_random that transforms an assignment $x \leftarrow$ iv_of_hash(r) when r is a fresh randomly generated value into $x \stackrel{R}{\leftarrow} IV$. By command crypto hash_to_iv_random, we apply this transformation: we replace the creation of IV_{CC} outlined above with the generation of a random value in IV.

    At this point, by command success, CryptoVerif is able to see that IV_{CC} is generated randomly and never used, and concludes that the secrecy of IV_{CC} is guaranteed.

    The secrecy can be proved for the other IV and keys by just repeating this last step for each one of them (possibly using the truncation functions for MAC or encryption keys instead of iv_of_hash).

4.5 About the Secrecy of Messages Sent in the Tunnel

In our model, we cannot prove the secrecy of messages sent in the tunnel. This point is actually related to known weaknesses in SSH with CBC mode encryption (which is still the only required encryption mode) [9, 4]. CBC mode encryption with chained IVs is not IND-CPA (indistinguishable under chosen plaintext attacks [8]), and this insecurity also applies to SSH [9]. This problem appears clearly when we try to do the proof. Because CryptoVerif does not allow encryption and decryption to generate random values internally or to maintain an internal state, even the interface of encryption in SSH differs from the one of IND-CPA encryption: in SSH, encryption receives a non-random IV while IND-CPA encryption receives random coins, and decryption receives an IV while IND-CPA decryption does not. Moreover, the oracle that decrypts the first block of a packet to get its length leaks the first four bytes of every packet. In fact, because of properties of CBC mode, using this oracle, one can compute the first four bytes of the cleartext of any ciphertext block [4, Section 3.2]. This problem is actually related to a real attack against some SSH implementations [4]: in practice, the length field is not immediately obtained by the adversary, but can be determined by sending messages block by block until one gets a reply, leading to the leakage of the cleartext. Such problems would be likely to remain unnoticed with an analysis of SSH in the symbolic model; that is why it is important to prove the protocol in the computational model.

In order to get a security proof, we could use counter mode encryption as specified in [10] instead of CBC mode encryption, by relying on its recent formalization in [42]. That would probably require extensions of CryptoVerif to keep a mutable counter internally. More generally, the main limitations of our approach come from limitations of CryptoVerif: it currently cannot handle mutable state, and may also be unable to prove some protocols secure even if they can be encoded. Additionally, it would also be interesting to formalize the SSH authentication and connection protocols.

4.6 Implementation

In order to implement the SSH Transport Layer Protocol, we wrote the network code and the cryptographic primitives. The cryptographic primitives are for the most part an interface to Cryptokit. Some specific algorithm encapsulations used by SSH had to be implemented. Message building and parsing are also implemented as if they were cryptographic primitives, with a basic specification of their properties: in particular, parsing is the inverse of message building. The network code sends and receives messages from the network, and also does some basic non-cryptographic manipulations (for instance, it sends the identification string directly).

We have verified that our client and server correctly interoperate with the server and client of OpenSSH. This shows that our implementation respects the message format and contents of SSH, and that it is a working implementation. However, we have omitted a few details of the SSH specification for simplicity: key re-exchange, IGNORE and DISCONNECT messages are not implemented yet. Since our compiler preserves security as shown in Part II, our implementation also satisfies the authentication of the server and the secrecy of session keys shown on the specification in Sections 4.3 and 4.4 (assuming the cryptographic primitives are correctly implemented). In order to give an idea on the amount of code this work represents, the CryptoVerif specification amounts to 331 lines of code, and we generate from it 531 lines of OCaml, split among multiple files. The manually written code representing the primitives and the authentication and connection protocols amount to 1124 lines.

The throughput of our implementation when tunneling random data on a local link is about 30 MB/s, whereas OpenSSH using the same algorithms as our implementation (those described in Section 4.1) ramps up to 90 MB/s on a Dual Core 3.2 GHz. It is slower because our generated code and the cryptographic primitives in Cryptokit are both slower than their OpenSSH equivalents, but it is still usable. We believe that the main reason for this slower speed is that our implementation allocates and copies strings when building messages instead of using a single buffer that would be modified in place. It would theoretically be possible to implement an optimizing compiler that would avoid string copies as much as possible, but the generated code would then be more difficult to relate to its CryptoVerif specification, and the compiler and its proof would be more complicated. The time required by our implementation to do an handshake (tunnel establishment and user authentication) varies wildly

depending on how we implement random number generation: much time may be spent waiting for entropy in the random number generator. OpenSSH uses the random number generator `arc4random` which uses an ARC4 pseudo-random generator regularly seeded with entropy gathered by the kernel, to reduce this waiting time to a minimum. However, the Cryptokit library does not provide access to `arc4random`, so one needs to seed a pseudo-random generator with new entropy at each run of SSH. This entropy can be taken from `/dev/random`, which waits until the kernel gathered enough entropy, so this is secure but slow, or from `/dev/urandom`, which does not wait, so this is fast, but may not be secure in case there is not enough entropy available. One could obviously extend Cryptokit to have access to `arc4random`. Ignoring the waiting time in the random number generator, the handshake on a local link for 2048 bit takes about 20 ms, both in our implementation and in OpenSSH. (This is the user plus system time, measured on an average of 100 runs.) This time is dominated by the Diffie-Hellman exponentiation and signature computation, which appears to be as fast in Cryptokit as in OpenSSH.

Conclusion

We presented in this part our compiler that translates an annotated CryptoVerif specification into an OCaml implementation. We applied this compiler to the SSH Transport Layer Protocol: we proved the authentication of the server and the secrecy of the session keys, and we generated an implementation of the protocol that could interact with an existing implementation of SSH, namely OpenSSH.

In order to ensure ourselves that our implementation is secure in the computational model, we need to prove that the compiler preserves security, which is the topic of Part II. This proof relies on assumptions on the implementation of cryptographic primitives and network code, as presented informally by Assumptions A1 to A6. In particular, in order to obtain a complete proof of security, we need to prove the security of the implementation of our cryptographic primitives, which is not done yet.

Part II

Proof of Correctness

Chapter 5

CryptoVerif Semantics

In this part, we begin by presenting the semantics of the input and output language of our compiler in Chapters 5 and 6. We also annotate the OCaml semantics in order to be able to carry the proof. Chapter 7 reviews the changes to the compiler in order to take into account these annotations. Finally, Chapter 8 presents the proof of correctness of the compiler.

This chapter presents the semantics of the CryptoVerif input language we described in Chapter 1.

5.1 Formal Semantics

We consider the language without the implementation annotations we described in Section 1.3. We adapt the semantics given in [16] for the channel front-end to the oracle front-end.

We present the semantics of the language in Figures 5.1, 5.2, and 5.3. The semantics is defined as a reduction relation on semantic configurations, which are tuples of the form $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$.

- The environment E is a mapping from array cells $x[\widetilde{a}]$ to their contents, where x is a variable, $\widetilde{a}$ gives the value of its replication indices, and the contents of $x[\widetilde{a}]$ is a bitstring value.
- The oracle body P is the oracle body currently running.
- The mapping $\mathcal{T}$ maps table names to their contents, which is the list of elements inserted in the table.
- The set $\mathcal{Q}$ contains the set of the callable oracle definitions.
- The list $\mathcal{R}$ is the call stack, which consists of triplets containing the variables with which the result should be bound and two oracle bodies, the first will be executed if the oracle returns a result with a return statement, and the second will be executed if the oracle terminates with an end statement.
- The list $\mathcal{E}$ is the list of events $e(a_1, \ldots, a_k)$ executed so far, by the construct event $e(M_1, \ldots, M_k)$.

Figure 5.1 Semantics (1)

Terms:

$$E, a \Downarrow a \tag{Cst}$$

$$2[mm]\frac{x[a_1, \ldots, a_m] \in \mathrm{Dom}(E)}{E, x[a_1, \ldots, a_m] \Downarrow E(x[a_1, \ldots, a_m])} \tag{Var}$$

$$\frac{\begin{array}{c}\forall j \leq m, E, M_j \Downarrow a_j \qquad f : T_1 \times \cdots \times T_m \to T \\ \forall j \leq m, a_j \in T_j\end{array}}{E, f(M_1, \ldots, M_m) \Downarrow f(a_1, \ldots, a_m)} \tag{Fun}$$

Oracle definitions:

$$\mathrm{reduce}(0) \stackrel{\mathrm{def}}{=} \emptyset \tag{Nil}$$

$$\mathrm{reduce}(Q_1 \mid Q_2) \stackrel{\mathrm{def}}{=} \mathrm{reduce}(Q_1) \cup \mathrm{reduce}(Q_2) \tag{Par}$$

$$\mathrm{reduce}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) \stackrel{\mathrm{def}}{=} \bigcup_{a=1}^{N} \mathrm{reduce}(Q\{a/i\}) \tag{Repl}$$

$$\begin{array}{c}\mathrm{reduce}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\mathrm{def}}{=} \\ \{(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P\}\end{array} \tag{Oracle}$$

During execution, terms may be reduced into constant bitstrings, so we add constant bitstrings a to the grammar of terms M. The notation $E, M \Downarrow a$ means that the term M evaluates to the bitstring a under the environment E. This relation is defined by rules (Cst), (Var), and (Fun) in Figure 5.1. The set $\mathrm{reduce}(Q)$, also defined in Figure 5.1, contains all oracle definitions provided by the oracle definition Q, with replication indices instantiated to all their possible values.

The function reduce′ defined in Chapter 3 is similar to the function reduce. In contrast to reduce, the function reduce′ returns an ordered list of oracles, without instantiating the replication indices.

The semantics is defined by probabilistic reduction rules between configurations: $\mathfrak{C} \to_p \mathfrak{C}'$ means that $\mathfrak{C}$ reduces into $\mathfrak{C}'$ with probability p. This relation is defined in Figures 5.2 and 5.3.

The rule (New) evaluates $x[\widetilde{a'}] \stackrel{R}{\leftarrow} T$ by choosing an element $a \in T$ and storing it in $E(x[\widetilde{a'}])$. The element $a \in T$ is chosen uniformly, so the probability of each choice is $1/|T|$ and this is possible only when T is a fixed-length type. The rule (Let) evaluates the term M and stores its value in $E(x[\widetilde{a'}])$. The rules (If1) and (If2) are straightforward.

The rules (Insert), (Get1), and (Get2) deal with tables of keys. The rule (Insert) evaluates the inserted element and adds it to the table Tbl, by adding it to the list $\mathcal{T}(Tbl)$. The rules (Get1) and (Get2) compute the list of elements that satisfy the condition of the get. When this list is empty, the else branch is taken by rule (Get2). When this list is not empty, the rule (Get1) chooses an element of this list l, stores it in $E(x_1[\widetilde{a'}]), \ldots, E(x_k[\widetilde{a'}])$, and takes the in branch.

Figure 5.2 Semantics (2)

Oracle bodies (1):

$$\frac{T \text{ fixed-length type} \qquad a \in T}{E, x[\widetilde{a'}] \overset{R}{\leftarrow} T; P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_{\frac{1}{|T|}} E[x[\widetilde{a'}] \mapsto a], P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}} \quad \text{(New)}$$

$$\frac{E, M \Downarrow a}{E, x[\widetilde{a'}] \leftarrow M; P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_1 E[x[\widetilde{a'}] \mapsto a], P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}} \quad \text{(Let)}$$

$$\frac{E, M \Downarrow \text{true}}{E, \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_1 E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}} \quad \text{(If1)}$$

$$\frac{E, M \Downarrow \text{false}}{E, \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}} \quad \text{(If2)}$$

$$\frac{\forall j \leq k, \quad E, M_j \Downarrow a_j}{\begin{array}{l} E, \mathsf{insert}\ Tbl(M_1, \ldots, M_k); P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_1 \\ \quad E, P, \mathcal{T}[Tbl \mapsto (a_1, \ldots, a_k) :: \mathcal{T}(Tbl)], \mathcal{Q}, \mathcal{R}, \mathcal{E} \end{array}} \quad \text{(Insert)}$$

$$\frac{\begin{array}{c} l = [(a_1, \ldots, a_k) \in \mathcal{T}(Tbl) \mid E[x_1[\widetilde{a'}] \mapsto a_1, \ldots, x_k[\widetilde{a'}] \mapsto a_k], M \Downarrow \text{true}] \\ (a_1^0, \ldots, a_k^0) \in l \\ S = \{1 \leq j \leq |l| \mid \mathsf{nth}(l, j) = (a_1^0, \ldots, a_k^0)\} \end{array}}{\begin{array}{l} E, \mathsf{get}\ Tbl(x_1[\widetilde{a'}], \ldots, x_k[\widetilde{a'}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \\ \quad \rightarrow_{\sum_{j \in S} \text{among}(\{1,\ldots,|l|\}, j)} \\ \quad\quad E[x_1[\widetilde{a'}] \mapsto a_1^0, \ldots, x_k[\widetilde{a'}] \mapsto a_k^0], P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \end{array}} \quad \text{(Get1)}$$

$$\frac{[(a_1, \ldots, a_k) \in \mathcal{T}(Tbl) \mid E[x_1[\widetilde{a'}] \mapsto a_1, \ldots, x_k[\widetilde{a'}] \mapsto a_k], M \Downarrow \text{true}] = [\,]}{\begin{array}{l} E, \mathsf{get}\ Tbl(x_1[\widetilde{a'}], \ldots, x_k[\widetilde{a'}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_1 \\ \quad E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \end{array}} \quad \text{(Get2)}$$

Figure 5.3 Semantics (3)

Oracle bodies (2):

$$\frac{\begin{array}{c}\forall i \leq l, E, M_i \Downarrow a'_i \qquad \widetilde{a'} = a'_1, \ldots, a'_l \qquad \forall j \leq k, E, N_j \Downarrow b_j \\ \exists x'_1, \ldots, x'_k, P'' \text{ such that} \\ Q_0 = (O[\widetilde{a'}](x'_1[\widetilde{a'}] : T'_1, \ldots, x'_k[\widetilde{a'}] : T'_k) := P'') \in \mathcal{Q} \\ E' = E[x'_1[\widetilde{a'}] \mapsto b_1, \ldots, x'_k[\widetilde{a'}] \mapsto b_k]\end{array}}{\begin{array}{l}E, \mathsf{let}\ (x_1[\widetilde{a}] : T_1, \ldots, x_{k'}[\widetilde{a}] : T_{k'}) = O[M_1, \ldots, M_l](N_1, \ldots, N_k) \\ \quad \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \quad \rightarrow_1 \\ \quad E', P'', \mathcal{T}, \mathcal{Q} \setminus \{Q_0\}, ((x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]), P, P') :: \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Call)}$$

$$\frac{\forall j \leq k, E, N_j \Downarrow b_j \qquad \mathcal{Q}' = \mathrm{reduce}(Q'')}{\begin{array}{l}E, \mathsf{return}(N_1, \ldots, N_k); Q'', \mathcal{Q}, ((x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), P, P') :: \mathcal{R}, \mathcal{E} \\ \rightarrow_1 E[x_1[\widetilde{a}] \mapsto b_1, \ldots, x_k[\widetilde{a}] \mapsto b_k], P, \mathcal{T}, \mathcal{Q} \cup \mathcal{Q}', \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Return)}$$

$$E, \mathsf{end}, \mathcal{T}, \mathcal{Q}, ((x_1[\widetilde{a}], \ldots, x_{k'}[\widetilde{a}]), P, P') :: \mathcal{R}, \mathcal{E} \rightarrow_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \quad \text{(End)}$$

$$\frac{\forall j \leq l, E, M_j \Downarrow a_j}{E, \mathsf{event}\ ev(M_1, \ldots, M_l); P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_1 E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, e(a_1, \ldots, a_l) :: \mathcal{E}} \quad \text{(Event)}$$

$$\frac{\begin{array}{c}\forall i \leq l, E, M_i \Downarrow a'_i \qquad E, N \Downarrow c \\ \text{the last replication above the definition of } O \text{ is } \mathsf{foreach}\ i_1 \leq N_1 \qquad a'_1 \leq N_1\end{array}}{\begin{array}{l}E, \mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[M_1, \ldots, M_l](N)\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \\ \quad \rightarrow_1 E, \\ \quad\ (\mathsf{let}\ (r'_{a'_1,r}[\widetilde{a}] : T, b_{a'_1,r}[\widetilde{a}] : bool) = O[a'_1, \ldots, a'_l](c)\ \mathsf{in} \\ \quad\quad \mathsf{if}\ b_{a'_1,r}[\widetilde{a}]\ \mathsf{then} \\ \quad\quad\quad (\mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[a'_1 + 1, a'_2, \ldots, a'_l](r'_{a'_1,r}[\widetilde{a}] : T) \\ \quad\quad\quad\ \mathsf{in}\ P\ \mathsf{else}\ P') \\ \quad\quad \mathsf{else}\ r[\widetilde{a}] \leftarrow r'_{a'_1,r}[\widetilde{a}]; P \\ \quad \mathsf{else}\ P'), \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Loop1)}$$

$$\frac{\begin{array}{c}\forall i \leq l, E, M_i \Downarrow a'_i \qquad E, N \Downarrow c \\ \text{the last replication above the definition of } O \text{ is } \mathsf{foreach}\ i_1 \leq N_1 \qquad a'_1 > N_1\end{array}}{\begin{array}{l}E, \mathsf{let}\ r[\widetilde{a}] : T = \mathsf{loop}\ O[M_1, \ldots, M_l](N)\ \mathsf{in}\ P\ \mathsf{else}\ P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \\ \quad \rightarrow_1 E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Loop2)}$$

The j-th element of the list l is chosen with probability among$(\{1,\ldots,|l|\},j)$, where among(S,b) is the probability that the element $b \in S$ is chosen among elements of the set S, according to an almost uniform distribution: we require that, for every set S, $\sum_{b\in S}$ among$(S,b) = 1$, among$(S,b) > 0$ for all $b \in S$, and $\sum_{b\in S}\left|\text{among}(S,b) - \frac{1}{|S|}\right| \leq \epsilon$ for some $\epsilon > 0$. Indeed, probabilistic Turing machines can choose random elements uniformly only in sets of cardinal a power of 2. For other sets, they can choose random elements with a probability distribution as close as we wish to uniform, that is, we can make ϵ as small as we wish in the formula above. In case the same element $a_1^0,\ldots,a_k^0$ occurs several times in the list l, the probability of choosing that element is the sum of the probabilities of all its occurrences. The probability of choosing $a_1^0,\ldots,a_k^0$ is then close to $m/|l|$, where m is the number of times this element appears in l.

The rule (Call) implements the oracle call $\mathsf{let}\ (x_1[\widetilde{a}] : T_1,\ldots,x_{k'}[\widetilde{a}] : T_{k'}) = O[M_1,\ldots,M_l](\underbrace{N_1,\ldots,N_k}) \ \mathsf{in}\ P\ \mathsf{else}\ P'$. It evaluates the indices $M_1,\ldots,M_l$ of the oracle to call into a' and its arguments $N_1,\ldots,N_k$ into $b_1,\ldots,b_k$; after evaluation, we want to call the oracle $O[\widetilde{a'}](b_1,\ldots,b_k)$. Then, it looks for the definition Q_0 of the oracle $O[\widetilde{a'}]$ in the callable oracles $\mathcal{Q}$. It calls Q_0 by removing it from the callable oracles, storing $b_1,\ldots,b_k$ in the arguments of Q_0, and running its body P''. The element $(x_1[\widetilde{a}],\ldots,x_{k'}[\widetilde{a}]),P,P')$ is pushed on the stack $\mathcal{R}$: $x_1[\widetilde{a}],\ldots,x_{k'}[\widetilde{a}]$ are the variables in which the return value of Q_0 should be stored, P is the process to execute when Q_0 returns, and P' is the process to execute when Q_0 terminates with end. The rule (Return) pops an element $((x_1[\widetilde{a}],\ldots,x_{k'}[\widetilde{a}]),P,P')$ from the stack, stores the return value in $x_1[\widetilde{a}],\ldots,x_{k'}[\widetilde{a}]$, and executes P. It adds to the set of callable oracles $\mathcal{Q}$ the oracles $\mathcal{Q}'$ defined in the oracle definition Q'' located after the return statement. The rule (End) also pops an element $((x_1[\widetilde{a}],\ldots,x_{k'}[\widetilde{a}]),P,P')$ from the stack, but executes the process P'. The rule (Event) adds the executed event to the list of events $\mathcal{E}$.

The rules (Loop1) and (Loop2) implement the loop statement. The rule (Loop1) performs one iteration of the loop. To that effect, it creates two fresh variable names $r'_{a'_1,r}$ and $b_{a'_1,r}$, calls the oracle O and stores its return values in these variables. When the boolean $b_{a'_1,r}[\widetilde{a}]$ returned by O is false, it ends the loop and continues by executing P with the result $r[\widetilde{a}]$ bound to the value of $r'_{a'_1,r}[\widetilde{a}]$. When $b_{a'_1,r}[\widetilde{a}]$ is true, it reruns the loop. If the oracle O terminates with an end statement, it ends the loop and continues by executing P'. The rule (Loop2) handles the case in which the loop stops by reaching the bound N_1 of the loop index.

The initial configuration for running the oracle definition Q is $\mathfrak{C}_{\mathrm{i}}(Q_0) \stackrel{\text{def}}{=} \emptyset, \mathsf{let}\ x[\,] : \mathit{bitstring} = O_{\mathrm{start}}()\ \mathsf{in}\ \mathsf{return}(x)\ \mathsf{else}\ \mathsf{end}, \mathcal{T}_0, \mathrm{reduce}(Q_0), \emptyset, [\,]$, where $\mathcal{T}_0(Tbl) = [\,]$ for all tables Tbl. This configuration starts by calling oracle O_{start}. The oracle definition Q_0 typically contains a protocol in parallel with an adversary.

Another interesting point is that, if a configuration $\mathfrak{C}$ reduces into another configuration, then the sum of the probabilities of all the possible reductions

from $\mathfrak{C}$ is 1:

$$\sum_{\{\mathfrak{C}' | \mathfrak{C} \rightarrow_{p(\mathfrak{C}')} \mathfrak{C}'\}} p(\mathfrak{C}') = 1 \,.$$

Definition 5.1 (Traces) *Let us denote traces with the symbol $\mathfrak{CT}$. A trace is a sequence of reductions $\mathfrak{CT} = \mathfrak{C}_0 \rightarrow_{p_1} \cdots \rightarrow_{p_n} \mathfrak{C}_n$ where $\mathfrak{C}_0, \ldots, \mathfrak{C}_n$ are semantic configurations such that $\mathfrak{C}_i \rightarrow_{p_{i+1}} \mathfrak{C}_{i+1}$ for $i = 0, \ldots, n-1$.*

A complete *trace is a trace such that there is no possible reduction from its last configuration.*

The probability of the trace $\mathfrak{CT}$ is $\Pr[\mathfrak{CT}] = p_1 \times \cdots \times p_n$. When the traces in a set of traces $\mathfrak{CTS}$ are not prefix of one another, the probability of $\mathfrak{CTS}$ is the sum of the probabilities of its elements.

The notation $\mathfrak{C} \rightarrow_p^* \mathfrak{C}'$ means that there exists a trace beginning at $\mathfrak{C}$ and ending at $\mathfrak{C}'$, and p is the probability of the set of all traces beginning at $\mathfrak{C}$ and stopping at their first occurrence of $\mathfrak{C}'$.

The notation $\mathfrak{C} \rightarrow_p^+ \mathfrak{C}'$ means that $\mathfrak{C} \rightarrow_p^* \mathfrak{C}'$ and $\mathfrak{C} \neq \mathfrak{C}'$, that is, all traces from $\mathfrak{C}$ to $\mathfrak{C}'$ have at least one step.

The notation $\mathfrak{C} \rightarrow^* \mathfrak{C}'$ means $\mathfrak{C} \rightarrow_1^* \mathfrak{C}'$. We denote the number of steps in the trace $\mathfrak{CT}$ as $|\mathfrak{CT}| = n$.

Intuitively, when traces in $\mathfrak{CTS}$ are not prefix of one another, they correspond to disjoint cases, so the probability of $\mathfrak{CTS}$ is the sum of probabilities of the traces in $\mathfrak{CTS}$. (When $\mathfrak{CT}$ is a prefix of $\mathfrak{CT}'$, the trace $\mathfrak{CT}'$ is a particular case of $\mathfrak{CT}$.)

In CryptoVerif, since for every reduction with a probabilistic choice, the environment E is modified so that we can determine from E which reduction was used, and one cannot remove elements from E, there will be at most one trace from one configuration to another. However, the notations of Definition 5.1 are also used for OCaml where there could be several configurations reducing to the same configuration, so they support this situation.

5.2 Oracle Unicity

Property 1.3 we presented in Chapter 1 guarantees that there exists a single callable definition for each oracle. This property is formalized by the following lemma.

Lemma 5.2 (Oracle name and indices unicity) *If the configuration $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ is reachable from the initial configuration $\mathfrak{C}_{\mathrm{i}}(Q_0)$ by reductions $\rightarrow_p$, then the set of callable oracles $\mathcal{Q}$ contains at most one oracle with a given name O and given replication indices $\widetilde{a}$.*

Proof As usual, a multiset S is defined as a function from elements to integers: $S(x)$ is the number of occurrences of x in the multiset S. Multiset union is defined as addition: $(S \uplus S')(x) = S(x) + S'(x)$. The maximum $\max(S, S')$ is the multiset such that $\max(S, S')(x) = \max(S(x), S'(x))$. The inclusion $S \subseteq S'$ is true when $\mathrm{Dom}(S) \subseteq \mathrm{Dom}(S')$ and $\forall x \in \mathrm{Dom}(S), S(x) \leq S'(x)$.

We define the multiset of available oracles inductively as follows:

$$
\begin{aligned}
&\mathit{Oracles}(0) = \emptyset\\
&\mathit{Oracles}(Q_1 \mid Q_2) = \mathit{Oracles}(Q_1) \uplus \mathit{Oracles}(Q_2)\\
&\mathit{Oracles}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) = \biguplus_{a \in [1,n]} \mathit{Oracles}(Q\{a/i\})\\
&\mathit{Oracles}(O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P) = \{O[\widetilde{a}]\} \uplus \mathit{Oracles}(P)\\
&\mathit{Oracles}(\mathsf{return}(M_1, \ldots, M_k); Q) = \mathit{Oracles}(Q)\\
&\mathit{Oracles}(\mathsf{end}) = \emptyset\\
&\mathit{Oracles}(x[\widetilde{a}] \stackrel{R}{\leftarrow} T; P) = \mathit{Oracles}(P)\\
&\mathit{Oracles}(x[\widetilde{a}] \leftarrow M; P) = \mathit{Oracles}(P)\\
&\mathit{Oracles}(\mathsf{insert}\ \mathit{Tbl}(M_1, \ldots, M_l); P) = \mathit{Oracles}(P)\\
&\mathit{Oracles}(\mathsf{get}\ \mathit{Tbl}(x_1[\widetilde{a}], \ldots, x_l[\widetilde{a}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P') =\\
&\qquad \max(\mathit{Oracles}(P), \mathit{Oracles}(P'))\\
&\mathit{Oracles}(\mathsf{event}\ e(M_1, \ldots, M_l); P) = \mathit{Oracles}(P)\\
&\mathit{Oracles}(\mathsf{let}\ (x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) = O[\widetilde{M}](\widetilde{M'})\ \mathsf{in}\ P\ \mathsf{else}\ P') =\\
&\qquad \max(\mathit{Oracles}(P), \mathit{Oracles}(P'))\\
&\mathit{Oracles}(\mathsf{let}\ x[\widetilde{i}] : T = \mathsf{loop}\ O[\widetilde{M}](M')\ \mathsf{in}\ P\ \mathsf{else}\ P') =\\
&\qquad \max(\mathit{Oracles}(P), \mathit{Oracles}(P'))\\
&\mathit{Oracles}(E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}) = \mathit{Oracles}(P) \uplus \biguplus_{Q \in \mathcal{Q}} \mathit{Oracles}(Q) \uplus\\
&\qquad\qquad \biguplus_{((x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), P', P'') \in \mathcal{R}} \max(\mathit{Oracles}(P'), \mathit{Oracles}(P''))
\end{aligned}
$$

We show that, for all configurations $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ reachable from the initial configuration $\mathfrak{C}_{\mathrm{i}}(Q_0)$, $\mathit{Oracles}(\mathfrak{C})$ contains no duplicates.

Let us first show this property for the initial configuration. We show by an easy induction on Q that $\biguplus_{Q' \in \mathrm{reduce}(Q)} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q)$. Therefore, by definition of $\mathfrak{C}_{\mathrm{i}}$, we have $\mathit{Oracles}(\mathfrak{C}_{\mathrm{i}}(Q_0)) = \biguplus_{Q' \in \mathrm{reduce}(Q_0)} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q_0)$. Next, we show by induction on Q_0 that $\mathit{Oracles}(Q_0)$ contains no duplicates.

- In the case $Q \mid Q'$, the oracles defined in Q and Q' are not in different branches of if or get, so by Property 1.3, they have different names. Hence, $\mathit{Oracles}(Q)$ and $\mathit{Oracles}(Q')$ do not both contain $O[\widetilde{a}]$ for the same O. We conclude that $\mathit{Oracles}(Q) \uplus \mathit{Oracles}(Q')$ contains no duplicates using the induction hypothesis.

- In the case $\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q$, by Property 1.3, the replication index i occurs as index in all definitions of oracles in Q, and in the same position. So the multisets $\mathit{Oracles}(Q\{a/i\})$ are disjoint for different choices of a. We conclude that $\biguplus_{a \in [1,n]} \mathit{Oracles}(Q\{a/i\})$ contains no duplicates using the induction hypothesis.

- In the case $O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P$, the definition of O is not in a branch of if or get different from P, so by Property 1.3, there is no definition of O in P. Hence $O[\widetilde{a}] \notin \mathit{Oracles}(P)$. We conclude that $\{O[\widetilde{a}]\} \uplus \mathit{Oracles}(P)$ contains no duplicates using the induction hypothesis.

- In all other cases, the result follows immediately from the induction hypothesis.

Furthermore, $\mathit{Oracles}(\mathfrak{C})$ decreases by reduction: if $\mathfrak{C} \rightarrow_p \mathfrak{C}'$, then we have $\mathit{Oracles}(\mathfrak{C}') \subseteq \mathit{Oracles}(\mathfrak{C})$. Indeed, the rules (New), (Let), (Insert), (Event), (Loop1) leave $\mathit{Oracles}(\mathfrak{C})$ unchanged. In the case of (Loop1), we use

$$\max(\max(\max(\mathit{Oracles}(P), \mathit{Oracles}(P')), \mathit{Oracles}(P)), \mathit{Oracles}(P')) = \max(\mathit{Oracles}(P), \mathit{Oracles}(P'))\,.$$

The rules (If1), (If2), (Get1), (Get2), (Loop2), (Return), (End) decrease the multiset $\mathit{Oracles}(\mathfrak{C})$ by replacing $\max(\mathit{Oracles}(P), \mathit{Oracles}(P'))$ with either $\mathit{Oracles}(P)$ or $\mathit{Oracles}(P')$. In the case of (Return), we also use $\biguplus_{Q' \in \mathrm{reduce}(Q'')} \mathit{Oracles}(Q') \subseteq \mathit{Oracles}(Q'')$. The rule (Call) removes the called oracle $O[\widetilde{a'}]$ from $\mathit{Oracles}(\mathfrak{C})$.

Therefore, for all configurations $\mathfrak{C}$ reachable from the initial configuration $\mathfrak{C}_\mathrm{i}(Q_0)$, $\mathit{Oracles}(\mathfrak{C})$ contains no duplicates.

By definition of $\mathit{Oracles}$, when $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$, we have $\{O[\widetilde{a}] \mid O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P \in \mathcal{Q}\} \subseteq \mathit{Oracles}(\mathfrak{C})$. (Both sides of the inclusion are multisets.) Therefore, $\mathcal{Q}$ contains at most one element $O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P$ for each $O[\widetilde{a}]$. □

This lemma proves that the rule (Call) is deterministic. Therefore, the rules (New) and (Get1) may make probabilistic choices; all other rules are deterministic.

5.3 Assumptions on the Language

For simplicity, we also assume the following points on the annotations of the process:

Assumption 5.3 *All oracle definitions are included in a role.*

Assumption 5.4 *No replication occurs above a parallel composition or a replication. When the definition of a role* role *is under replication* foreach $i \leq n$ do role$\{Q$*, its contents Q consists of an oracle definition $O[\widetilde{i}](\ldots) := \ldots$ or of a parallel composition of such oracle definitions (without replication).*

A process can be transformed so that no replication occurs above a parallel composition by distributing the replications into the parallel compositions: foreach $i \leq n$ do $(P_1 \mid P_2)$ can be transformed into (foreach $i_1 \leq n_1$ do P_1) | (foreach $i_2 \leq n_2$ do P_2). We can encode nested replications by adding a dummy oracle between the two replications: the process foreach $i \leq n$ do foreach $j \leq$

n' do P can be transformed into foreach $i \leq n$ do $O() :=$ return(); foreach $j \leq n'$ do P.

By Properties 1.3, 1.4, and 1.5, there cannot be, in the same process, a definition of an oracle O directly under replication and another definition of the same oracle O not directly under replication. Hence, we can use the phrase "O is under replication" unambiguously. Moreover, by Property 1.4, the bound of the replication above a definition of an oracle O is the same for all definitions of O.

Assumption 5.5 *For each oracle O under replication, we let N_O be the bound of the replication above the definition of O. For each role* role *under replication, we let N_{role} be the bound of the replication above the definition of* role*. All these bounds N_O and N_{role} are pairwise distinct.*

This assumption allows us to be more precise when counting the number of times an oracle has been called, by using a distinct bound for each oracle.

These assumptions are relaxed in our implementation.

Chapter 6

OCaml Semantics

This section presents the semantics of OCaml, the target language of our compiler we presented in Chapter 2. To define the formal semantics, we adapted the semantics by Scott Owens et al. [41]. This semantics is a small step operational semantics of the core part of the OCaml language. We modified it in several ways.

A security protocol typically involves several programs running in parallel on different machines. We model this situation by considering several threads. To manage threads, we introduce two new expressions, $\mathsf{addthread}(\mathit{program})$ and $\mathsf{schedule}(e)$. The expression $\mathsf{addthread}(\mathit{program})$ creates a new thread that runs the program *program*. The expression $\mathsf{schedule}(e)$ stops execution of the current thread and continues execution of the thread number e. (Threads are designated by integer numbers. The initial thread, started at the beginning of the program, has number 1. The threads created by subsequent calls to addthread have numbers starting at 2 and increasing by one each time a new thread is created.)

6.1 Formal Semantics

We define step by step the semantics of the various constructs of the language.

6.1.1 Pattern matching

We define the predicate matches in Figure 6.1: we have v matches $\mathit{pat} \triangleright \mathit{env}$ when the value v matches the pattern *pat*, and the environment *env* is a mapping from the variables of *pat* to their values, computed by the pattern matching. The operation $\mathit{env} \oplus \mathit{env}' \stackrel{\text{def}}{=} \mathit{env}_{|\overline{\mathrm{Dom}(\mathit{env}')}} \cup \mathit{env}'$ adds the bindings of env' to those of *env*; when a variable is bound in both environments, the binding of env' is kept. Since patterns are linear, in Figure 6.1, the operation $\mathit{env} \oplus \mathit{env}'$ is always used with environments *env* and env' that have disjoint domains; the general case is used below. We also define v matches *pat* as $\exists \mathit{env}, v$ matches $\mathit{pat} \triangleright \mathit{env}$.

6.1.2 Primitives

The semantics of primitives is defined in Figure 6.2. This semantics is defined by rules of the form $\mathit{prim}\ v_1\ \ldots\ v_n \xrightarrow{L}_p e$ where *prim* is an n-ary primitive.

Figure 6.1 Matches predicate

$$v \text{ matches } x \triangleright \{x \mapsto v\} \qquad \text{(Variable)}$$

$$v \text{ matches } \_ \triangleright \emptyset \qquad \text{(Any)}$$

$$\frac{\forall 1 \leq i \leq n, v_i \text{ matches } pat_i \triangleright env_i}{(v_1, \ldots, v_n) \text{ matches } (pat_1, \ldots, pat_n) \triangleright \bigoplus_{i=1}^{n} env_i} \qquad \text{(Tuple)}$$

$$\frac{v_1 \text{ matches } pat_1 \triangleright env_1 \qquad v_2 \text{ matches } pat_2 \triangleright env_2}{v_1 :: v_2 \text{ matches } pat_1 :: pat_2 \triangleright env_1 \oplus env_2} \qquad \text{(List)}$$

Such a rule means that $prim\ v_1\ \ldots\ v_n$ reduces to e with probability p. In contrast to [41], the semantics is probabilistic, because of the presence of the primitive random. The probability p is omitted when it is 1. The label L is used to reflect the operations on locations. It is empty when locations are unaffected. The label ref $v = l$ means that a new location l is created, with contents v. The label $!l = v$ means that the current contents of location l is v. The label $l := v$ means that the contents of the location l is changed into v. The rules are straightforward; they reflect the semantics defined informally in Chapter 2. One is not allowed to test equality between functional values, so we use the predicate funval, also defined in Figure 6.2, to test whether a value is functional, and raise the exception Invalid_argument when we try to test equality between functional values. There is no rule for the primitive raise: raise v is an exceptional value, it does not reduce.

6.1.3 Expressions and Programs

The semantics of [41] substitutes variables with their values. Instead, we define an environment *env* that maps variables to their values. This way, it is easier to relate the OCaml state to the CryptoVerif state which also contains an environment. Because of this change, we also need to add an explicit call stack *stack*. The stack is a list of pairs (env, C), where C is a minimal *evaluation context*, that is, an expression with a hole $[\cdot]$, such that the expression inside the hole can be immediately evaluated. We define a minimal evaluation context as:

$C_m ::=$	minimal expression evaluation context
$e\ [\cdot]$	apply
$[\cdot]\ v$	apply function
let $pat = [\cdot]$ in e	let
$[\cdot]; e$	sequence
if $[\cdot]$ then e_1 else e_2	if
match $[\cdot]$ with pm	match
try $[\cdot]$ with pm	try
$(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)$	tuple
$e :: [\cdot]$	cons1
$[\cdot] :: v$	cons2

Figure 6.2 Rules for OCaml primitives

Functional values:

$$\frac{prim \text{ is an } n\text{-ary primitive} \qquad 0 \le j < n}{\mathsf{funval}(prim\ v_1\ \ldots\ v_j)} \quad \text{(Primitive)}$$

$$\mathsf{funval}(\mathsf{function}[env, pm]) \quad \text{(Function)}$$

$$\mathsf{funval}(\mathsf{letrec}[env, \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}\ \mathsf{in}\ x_i]) \quad \text{(Let rec)}$$

Primitives:

$$\mathsf{not}\ \begin{cases} \mathsf{not}\ \mathrm{false} \longrightarrow \mathrm{true} & \text{(Not1)} \\ \mathsf{not}\ \mathrm{true} \longrightarrow \mathrm{false} & \text{(Not2)} \end{cases}$$

$$(=)\ \begin{cases} \dfrac{\mathsf{funval}(v) \text{ or } \mathsf{funval}(v')}{v = v' \longrightarrow \mathsf{raise\ Invalid\_argument}} & \text{(Funval)} \\ c = c \longrightarrow \mathrm{true} & \text{(Constant1)} \\ \dfrac{c \neq c'}{c = c' \longrightarrow \mathrm{false}} & \text{(Constant2)} \\ v_1 :: v_2 = v_1' :: v_2' \longrightarrow v_1 = v_1' \,\&\&\, v_2 = v_2' & \text{(List1)} \\ v_1 :: v_2 = [\,] \longrightarrow \mathrm{false} & \text{(List2)} \\ [\,] = v_1' :: v_2' \longrightarrow \mathrm{false} & \text{(List3)} \\ (v_1, \ldots, v_n) = (v_1', \ldots, v_n') \longrightarrow v_1 = v_1' \,\&\&\, \ldots \,\&\&\, v_n = v_n' & \text{(Tuples)} \end{cases}$$

$$\mathsf{ref}\ \begin{cases} \mathsf{ref}\ v \xrightarrow{\mathsf{ref}\ v=l} l & \text{(New ref)} \end{cases}$$

$$(:=)\ \begin{cases} l := v \xrightarrow{l:=v} () & \text{(Assign)} \end{cases}$$

$$!\ \begin{cases} !l \xrightarrow{!l=v} v & \text{(Dereference)} \end{cases}$$

$$\mathsf{random}\ \begin{cases} \dfrac{a \in \{\mathrm{true}, \mathrm{false}\}}{\mathsf{random}\ () \longrightarrow_{1/2} a} & \text{(Random)} \end{cases}$$

$$
\begin{array}{lll}
& \mathsf{schedule}([\cdot]) & \text{schedule} \\
C_{pm} ::= & & \text{minimal program evaluation context} \\
& \mathsf{let}\ pat = [\cdot];;\ definitions & \text{let}
\end{array}
$$

For instance, $e\ [\cdot]$ and $[\cdot]\ v$ are evaluation contexts, so we evaluate the argument of applications first, and when it becomes a value, we evaluate the function. We denote by $C_m[e]$ the context C_m with the hole $[\cdot]$ replaced by e, and similarly for C_{pm}. The stack contains a minimal program evaluation context C_{pm} in the last element of the list and expression evaluation contexts C_m in the other elements if it is non-empty.

Hence, we evaluate expressions and programs by reducing triples $env, pe, stack$, where pe means program *program* or expression e. The reduction rules $env, pe, stack \xrightarrow{L}_p env', pe', stack'$ are defined in Figures 6.3 and 6.4 for expressions and Figure 6.5 for programs. The label L is defined above in Section 6.1.2. These reductions are probabilistic; the probability p is omitted when it is 1. Most rules are straightforward. In order to evaluate an expression $C_m[e]$, we need to reduce e under the context C_m. To do that, we push the context C_m on the stack with the current environment by rule (Context in), evaluate the expression e until it becomes a value v, and finally pop the context C_m from the stack by rule (Context out), inserting the obtained value v in C_m, yielding $C_m[v]$. In case the expression e raises an exception v, we use rules (Context raise1) and (Context raise2). If the context C_m is not a try context, the result of $C_m[e]$ is also raise v by (Context raise1). If C_m is a try context, we evaluate that try by (Context raise2), followed by (Try2). The rules (Let ctx in), (Let ctx out), and (Let ctx raise) play the same role as (Context in), (Context out), and (Context raise1) respectively, for programs instead of expressions: they allow reducing under the minimal program evaluation context $\mathsf{let}\ pat = [\cdot];;\ definitions$. There is no rule corresponding to (Context raise2) for programs because there is no try program context.

Example 10 Let us present as an example the reduction of a simple program in an empty environment and an empty stack:

$$\emptyset, \mathsf{let}\ x = \mathsf{if}\ \mathsf{random}\ ()\ \mathsf{then}\ 0\ \mathsf{else}\ 1;;, [\,]\,.$$

We first reduce the expression part of the let, by keeping in the stack the fact that the expression is under the context $\mathsf{let}\ x = [\cdot]$. This expression reduces eventually to a value, and at this point we plug this value back into the context. So we first reduce the previous configuration by (Let ctx in) into:

$$\emptyset, \mathsf{if}\ \mathsf{random}\ ()\ \mathsf{then}\ 0\ \mathsf{else}\ 1, [\emptyset, \mathsf{let}\ x = [\cdot];;]$$

By (Context in), we prepare to reduce the condition of the if:

$$\emptyset, \mathsf{random}\ (), [\emptyset, \mathsf{if}\ [\cdot]\ \mathsf{then}\ 0\ \mathsf{else}\ 1; \emptyset, \mathsf{let}\ x = [\cdot];;] \tag{6.1}$$

By (Random), random () reduces to true with probability 1/2 and false with probability 1/2. For the purpose of the example, let us consider the case where

random () reduces to true. By (Primitives), the configuration reduces with probability 1/2 into

$$\emptyset, \text{true}, [\emptyset, \mathsf{if}\ [\cdot]\ \mathsf{then}\ 0\ \mathsf{else}\ 1; \emptyset, \mathsf{let}\ x = [\cdot];;\,]$$

By (Context out), we insert the value of the condition back into the if:

$$\emptyset, \mathsf{if}\ \text{true}\ \mathsf{then}\ 0\ \mathsf{else}\ 1, [\mathsf{let}\ x = [\cdot];;\,]$$

By (If1), we evaluate the if:

$$\emptyset, 0, [\mathsf{let}\ x = [\cdot];;\,]$$

By (Let ctx out), we insert the obtained value back into the context let $x = [\cdot];;$

$$\emptyset, \mathsf{let}\ x = 0;;\,, [\,]$$

By (Variable), we have that 0 matches $x \triangleright \{x \mapsto 0\}$. So, by (Let match1), the configuration reduces into the following last configuration:

$$\{x \mapsto 0\}, \varepsilon, [\,]$$

The configuration of Equation 6.1 reduces similarly in the case where random reduces to false into the last configuration $\{x \mapsto 1\}, \varepsilon, [\,]$. □

The expressions addthread(*program*) and schedule(e) are treated specially because they alter parts of the semantic configuration other than $env, pe, stack$. Their treatment is detailed in Section 6.1.5.

6.1.4 Store

As usual, the contents of locations are stored in a *store*, which maps locations to their current values. Figure 6.6 defines the relation $store \xrightarrow{L} store'$. If a program or an expression reduces by $env, pe, stack \xrightarrow{L}_p env', pe', stack'$, then the store *store* will be updated into *store*′ such that $store \xrightarrow{L} store'$. When L is empty, the store is unchanged by rule (Store empty). When L is $!l = v$, the store is also unchanged, but the reduction succeeds only when the contents of l is v, by rule (Store lookup). When L is $l := v$, the store is updated so that l contains v, by rule (Store assign). When L is ref $v = l$, a new location l is created with contents v, so the contents of l must not be defined in the initial store, by rule (Store alloc).

6.1.5 Toplevel Reduction

In contrast to [41], we consider several threads running in parallel. Each thread has a configuration $Th_i = \langle env_i, pe_i, stack_i, store_i \rangle$ that contains the current $env_i, pe_i, stack_i$ as explained in Section 6.1.3, as well as the contents of locations local to this thread, in a store $store_i$, as explained in Section 6.1.4. The complete semantic configuration is then

$$\mathcal{C} = [Th_1, \ldots, Th_n], globalstore, tj$$

Figure 6.3 Rules for expressions

$$env, x, stack \longrightarrow env, env(x), stack \qquad \text{(Env)}$$

$$\frac{prim\ v_1\ \ldots\ v_n \xrightarrow{L}_p e}{env, prim\ v_1\ \ldots\ v_n, stack \xrightarrow{L}_p env, e, stack} \qquad \text{(Primitives)}$$

$$\frac{\begin{array}{c}e \text{ is not a value} \\ \text{and, when } C_m \text{ is a } \mathsf{try} \text{ context, } e \text{ is not an exceptional value}\end{array}}{env, C_m[e], stack \longrightarrow env, e, (env, C_m) :: stack} \qquad \text{(Context in)}$$

$$env', v, (env, C_m) :: stack \longrightarrow env, C_m[v], stack \qquad \text{(Context out)}$$

$$\frac{C_m \text{ is not a } \mathsf{try} \text{ context}}{env', \mathsf{raise}\ v, (env, C_m) :: stack \longrightarrow env, \mathsf{raise}\ v, stack} \qquad \text{(Context raise1)}$$

$$\frac{C_m \text{ is a } \mathsf{try} \text{ context}}{env', \mathsf{raise}\ v, (env, C_m) :: stack \longrightarrow env, C_m[\mathsf{raise}\ v], stack} \qquad \text{(Context raise2)}$$

$$env, \mathsf{function}\ pm, stack \longrightarrow env, \mathsf{function}[env, pm], stack \qquad \text{(Closure)}$$

$$env, \mathsf{function}[env', pm]\ v_0, stack \longrightarrow env', \mathsf{match}\ v_0\ \mathsf{with}\ pm, stack \qquad \text{(Expr apply)}$$

$$env, v; e, stack \longrightarrow env, e, stack \qquad \text{(Sequence)}$$

$$env, \mathsf{if}\ \text{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, stack \longrightarrow env, e_1, stack \qquad \text{(If1)}$$

$$env, \mathsf{if}\ \text{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, stack \longrightarrow env, e_2, stack \qquad \text{(If2)}$$

$$\frac{v\ \mathsf{matches}\ pat \triangleright env'}{\begin{array}{l}env, \mathsf{match}\ v\ \mathsf{with}\ pat \to e \mid pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n, stack \longrightarrow \\ env \oplus env', e, stack\end{array}} \qquad \text{(Match1)}$$

$$\frac{\neg(v\ \mathsf{matches}\ pat)}{\begin{array}{l}env, \mathsf{match}\ v\ \mathsf{with}\ pat \to e \mid pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n, stack \longrightarrow \\ env, \mathsf{match}\ v\ \mathsf{with}\ pat_1 \to e_1 \mid \ldots \mid pat_n \to e_n, stack\end{array}} \qquad \text{(Match2)}$$

$$\frac{\neg(v\ \mathsf{matches}\ pat)}{env, \mathsf{match}\ v\ \mathsf{with}\ pat \to e, stack \longrightarrow env, \mathsf{raise\ Match\_failure}, stack} \qquad \text{(Match fail)}$$

$$env, \mathsf{try}\ v\ \mathsf{with}\ pm, stack \longrightarrow env, v, stack \qquad \text{(Try1)}$$

$$env, \mathsf{try\ raise}\ v\ \mathsf{with}\ pm, stack \longrightarrow env, \mathsf{match}\ v\ \mathsf{with}\ pm \mid \_ \to \mathsf{raise}\ v, stack \qquad \text{(Try2)}$$

Figure 6.4 Rules for expressions (continued)

$$\frac{v \text{ matches } pat \triangleright env'}{env, \mathsf{let}\ pat = v\ \mathsf{in}\ e, stack \longrightarrow env \oplus env', e, stack} \quad \text{(Let1)}$$

$$\frac{\neg(v \text{ matches } pat)}{env, \mathsf{let}\ pat = v\ \mathsf{in}\ e, stack \longrightarrow env, \mathsf{raise\ Match\_failure}, stack} \quad \text{(Let2)}$$

$$\frac{letrecenv = \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}}{\begin{array}{l} env, \mathsf{let\ rec}\ x_1 = \mathsf{function}\ pm_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ x_n = \mathsf{function}\ pm_n\ \mathsf{in}\ e, stack \\ \quad \longrightarrow env[x_1 \mapsto \mathsf{letrec}[env, letrecenv\ \mathsf{in}\ x_1], \\ \qquad \ldots, \\ \qquad x_n \mapsto \mathsf{letrec}[env, letrecenv\ \mathsf{in}\ x_n]], e, stack \end{array}}$$

(Closure let rec)

$$\frac{letrecenv = \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}}{\begin{array}{l} env, \mathsf{letrec}[env', letrecenv\ \mathsf{in}\ x_i]\ v_0, stack \longrightarrow \\ \quad env'[x_1 \mapsto \mathsf{letrec}[env', letrecenv\ \mathsf{in}\ x_1], \\ \qquad \ldots, \\ \qquad x_n \mapsto \mathsf{letrec}[env', letrecenv\ \mathsf{in}\ x_n]], \\ \quad \mathsf{match}\ v_0\ \mathsf{with}\ pm_i, stack \end{array}} \quad \text{(Expr letrec apply)}$$

Figure 6.5 Rules for programs

$$\frac{e \text{ is not a value}}{env, \mathsf{let}\ pat = e;; definitions, [\,] \longrightarrow env, e, [env, \mathsf{let}\ pat = [\cdot];; definitions]} \quad \text{(Let ctx in)}$$

$$env', v, [env, \mathsf{let}\ pat = [\cdot];; definitions] \longrightarrow env, \mathsf{let}\ pat = v;; definitions, [\,] \quad \text{(Let ctx out)}$$

$$env', \mathsf{raise}\ v, [env, \mathsf{let}\ pat = [\cdot];; definitions] \longrightarrow env, \mathsf{raise}\ v, [\,] \quad \text{(Let ctx raise)}$$

$$\frac{v \text{ matches } pat \triangleright env'}{env, \mathsf{let}\ pat = v;; definitions, [\,] \longrightarrow env \oplus env', definitions, [\,]} \quad \text{(Let match1)}$$

$$\frac{\neg(v \text{ matches } pat)}{env, \mathsf{let}\ pat = v;; definitions, [\,] \longrightarrow env, \mathsf{raise\ Match\_failure}, [\,]} \quad \text{(Let match2)}$$

$$\frac{letrecenv = \{x_1 \mapsto \mathsf{function}\ pm_1, \ldots, x_n \mapsto \mathsf{function}\ pm_n\}}{\begin{array}{l} env, \mathsf{let\ rec}\ x_1 = \mathsf{function}\ pm_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ x_n = \mathsf{function}\ pm_n;; definitions, [\,] \\ \quad \longrightarrow env[x_1 \mapsto \mathsf{letrec}[env, letrecenv\ \mathsf{in}\ x_1], \\ \qquad \ldots, \\ \qquad x_n \mapsto \mathsf{letrec}[env, letrecenv\ \mathsf{in}\ x_n]], definitions, [\,] \end{array}}$$

(Closure let rec)

Figure 6.6 Store rules

$$store \longrightarrow store \qquad \text{(Store empty)}$$

$$\frac{store(l) = v}{store \xrightarrow{!l=v} store} \qquad \text{(Store lookup)}$$

$$\frac{l \in \mathrm{Dom}(store)}{store \xrightarrow{l:=v} store[l \mapsto v]} \qquad \text{(Store assign)}$$

$$\frac{l \notin \mathrm{Dom}(store)}{store \xrightarrow{\mathsf{ref}\ v=l} store[l \mapsto v]} \qquad \text{(Store alloc)}$$

where tj is the number of the thread currently being executed, and *globalstore* is a store for locations shared between threads. We use it to model the communication between threads by storing messages in global locations, and to store the files containing private data from the CryptoVerif process (free variables of roles and tables). In practice, these files may be copied from one machine to another by the user, so they are actually shared between several threads. The values in the global store contain no closure and no reference. (In OCaml, closures and references can be written to a file only by marshalling, but marshalling is ruled out by Assumption A4, since it may bypass the type system.) The global store contains locations in a set S_g, while the local stores contain locations in an infinite set S_l, with $S_g \cap S_l = \emptyset$.

The reduction rules for semantic configurations $\mathcal{C}$ are defined in Figure 6.7. Actually, this figure defines three relations. The relation $Th \rightarrow_p Th'$, defined by rule (Thread), handles all operations that deal with the current thread only. It updates the store using the same label L as the one used for evaluating the program or the expression, and it checks that this label concerns the local store of the thread. (The location l, if any, must be in S_l.)

Second, the relation $Th, globalstore \rightarrow_p Th', globalstore'$, defined by rules (Globalstore1) and (Globalstore2), handles all operations local to one thread and the global store operations. By rule (Globalstore1), it uses the relation $Th \rightarrow_p Th'$ to handle the operations local to one thread. By rule (Globalstore2), it handles the global store operations. It updates the global store using the same label L as the one used for evaluating the program or the expression, and it checks that this label concerns the global store. The location l must be in S_g, and the creation of a location in the global store is forbidden. (Otherwise, one would need a way to tell the system whether a new location should be created in the local or in the global store, and to communicate the global locations to the other threads.) We assume that all locations of the global store are initialized at the beginning of the program.

Finally, the relation $\mathcal{C} \rightarrow_p \mathcal{C}'$, defined by the last four rules of Figure 6.7, gives the semantics of the full language. Rule (Toplevel) runs the current thread tj, using the relation $Th, globalstore \rightarrow_p Th', globalstore'$. Rule (Toplevel add thread) defines the semantics of $\mathsf{addthread}(program)$: it creates a new thread that

runs the program *program*, with empty environment, stack, and store. Rules (Toplevel schedule1) and (Toplevel schedule2) define the semantics of schedule: schedule(tj') schedules thread number tj' when this thread exists, and otherwise it raises the exception Invalid_argument.

Splitting the definition of the semantics into three relations allows us to lighten notations in proofs: we can use the reduction on a thread, or on a thread and the global store, without mentioning the other components when they are not affected.

The construct addthread does not allow using the same local store in several threads, which corresponds to forbidding fork in the middle of a role. Moreover, we reduce only the active thread, and we change threads only with schedule. So we can only change threads in code defined by the adversary, because neither the primitives nor the generated modules use schedule. So a call to an oracle cannot be interleaved with other threads. This property corresponds to Assumption A6: if several oracles cannot interleave reads and writes in the same table file, one can reconstruct a well-defined call order for these oracles in the CryptoVerif process, which processes one oracle call after another, so that the calls can be simulated in our semantics.

6.1.6 Modules

OCaml programs typically contain several modules. We adopt a very simple model of modules. A module named μ consists of an OCaml program $program(\mu)$ and its interface $interface(\mu)$ that is the set of OCaml identifiers defined in μ and usable in other modules. When needed to distinguish identifiers coming from different modules, we use identifiers of the form $\mu.x$ for variables defined in module μ. A correct OCaml program is then of the form $program = program(\mu_1);; \ldots ;; program(\mu_n);;$, where, for all $i \leq n$, the free variables of μ_i are defined in the interfaces of μ_j with $j < i$.

Such a program is run by using the previous reduction rules from the initial configuration

$$\mathcal{C}_0(program) = [\langle \emptyset, program, [\,], \emptyset \rangle], globalstore_0, 1$$

where $globalstore_0 = \{l \mapsto initval_l \mid l \in S_g\}$ is the initial value of the global store, and $initval_l$ is the default value for location l: the empty list [ ] for lists, the empty string "" for strings, 0 for integers, false for booleans. (Each location implicitly has a type. Values in the global store cannot contain locations and closures, so we do not define a default value for them.) The program *program* does not contain closures nor locations in S_l, but may contain locations in S_g. (Closures are created by function and letrec; locations in S_l are created by ref.)

Although we ignore types is our syntax, we suppose that our OCaml programs are well-typed, which is checked by the OCaml compiler, and we use the guarantee that well-typed programs do not go wrong: a program stops only when the current thread has been reduced into the empty definition list or an exception raise v (with the empty stack).

Figure 6.7 Top level rules

$$\frac{\begin{array}{l} store \xrightarrow{L} store' \\ L \text{ is empty or } L \text{ is } !l = v,\ l := v, \text{ or } \mathsf{ref}\ v = l \text{ with } l \in S_l \\ env, pe, stack \xrightarrow{L}_p env', pe', stack' \end{array}}{\langle env, pe, stack, store\rangle \longrightarrow_p \langle env', pe', stack', store'\rangle} \quad \text{(Thread)}$$

$$\frac{Th \longrightarrow_p Th'}{Th, globalstore \longrightarrow_p Th', globalstore} \quad \text{(Globalstore1)}$$

$$\frac{\begin{array}{l} globalstore \xrightarrow{L} globalstore' \\ L \text{ is } !l = v \text{ or } l := v \text{ with } l \in S_g \\ env, pe, stack \xrightarrow{L}_p env', pe', stack' \end{array}}{\langle env, pe, stack, store\rangle, globalstore \longrightarrow_p \langle env', pe', stack', store\rangle, globalstore'} \quad \text{(Globalstore2)}$$

$$\frac{Th, globalstore \longrightarrow_p Th', globalstore'}{\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, Th, Th_{tj+1}, \ldots, Th_n], globalstore, tj \longrightarrow_p \\ [Th_1, \ldots, Th_{tj-1}, Th', Th_{tj+1}, \ldots, Th_n], globalstore', tj\end{array}} \quad \text{(Toplevel)}$$

$$\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{addthread}(program), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n, \langle \emptyset, program, [\,], \emptyset\rangle], \\ \quad globalstore, tj\end{array} \quad \text{(Toplevel add thread)}$$

$$\frac{1 \leq tj' \leq n}{\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{schedule}(tj'), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], globalstore, tj'\end{array}} \quad \text{(Toplevel schedule1)}$$

$$\frac{tj' < 1 \text{ or } tj' > n}{\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{schedule}(tj'), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{raise\ Invalid\_argument}, stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj\end{array}} \quad \text{(Toplevel schedule2)}$$

6.1.7 Equivalence Modulo Renaming of Locations

The rule (Store alloc) is non-deterministic, since the new location l can be any unused location in S_l. To remove this non-determinism, we consider equivalence classes of OCaml semantic configurations modulo renaming of locations in S_l. We still denote these equivalence classes as OCaml configurations $\mathcal{C}$, and denote an equivalence class by one of its members. On these equivalence classes, the semantics is purely probabilistic. (There is no non-deterministic choice.) If a configuration $\mathcal{C}$ can reduce, then the sum of the probabilities of all possible reductions is 1:

$$\sum_{\{\mathcal{C}' | \mathcal{C} \to_{p(\mathcal{C}')} \mathcal{C}'\}} p(\mathcal{C}') = 1$$

Moreover, for each reduction $\mathcal{C} \to_p \mathcal{C}'$, we have $p > 0$.

We will also use notations similar to Definition 5.1 for the OCaml semantics. We denote by $\mathcal{CT}$ an OCaml trace, $\mathcal{CTS}$ a set of OCaml traces, and we also use the notation $\to^*$ for reductions with several steps.

6.2 Instrumentation

In order to prove the correctness of our compiler, we instrument OCaml code in three ways; this section details this instrumentation and proves that it does not alter the semantics of OCaml.

First, we add a new kind of functions and closures tagfunction that behave exactly in the same way as regular functions and closures, but are labeled with additional tags. We use these tagged functions to differentiate functions coming from our generated code and functions coming from the adversary. Hence, we add two new expressions $\mathsf{tagfunction}^t\ pm$ for tagged functions and $\mathsf{tagfunction}^{t,\tau}[env, pm]$ for the corresponding closures. We also add $\mathsf{tagfunction}^{t,\tau}[env, pm]$ to the values. The tag t indicates the origin of the function or closure; it will be an oracle name or a role name, indicating that the function implements this oracle or role. The tag τ is a fresh tag generated when the function is reduced into a closure: each new closure gets a different tag, so that two closures are the same if and only if they have the same tag. This property will be used in Section 8.3 to count the number of calls to the same closure. The semantic rules for tagged functions are given in Figure 6.8. They are the same as those for ordinary functions, except for the addition of tags. Much like for locations, we consider traces modulo renaming of tags τ, so that the choice of a fresh tag τ in (Tagged closure) does not lead to non-determinism. The condition that τ is fresh in this rule means that τ is distinct from all tags previously used in the considered trace.

Second, we need to be able to match CryptoVerif events, so we add to the semantic configuration an element *events* that contains the list of the events executed until now. We add the expression $\mathsf{event}\ ev(e_1, \ldots, e_k)$ that adds the event $ev(v_1, \ldots, v_k)$ to *events* when $e_1, \ldots, e_k$ evaluate to the values $v_1, \ldots, v_k$ respectively. We consider a new minimal expression evaluation context $\mathsf{event}(ev(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n))$, for evaluating the arguments of events. Events serve in specifying security properties of protocols, so they appear in generated code, but cannot be used by the adversary.

Figure 6.8 Semantics of tagged functions

$$\frac{}{\mathsf{funval}(\mathsf{tagfunction}^{t,\tau}[env, pm])} \quad \text{(Tagged funval)}$$

$$\frac{\tau \text{ fresh}}{env, \mathsf{tagfunction}^{t}\ pm, stack \longrightarrow env, \mathsf{tagfunction}^{t,\tau}[env, pm], stack} \quad \text{(Tagged closure)}$$

$$env, \mathsf{tagfunction}^{t,\tau}[env', pm]\ v_0, stack \longrightarrow env', \mathsf{match}\ v_0\ \mathsf{with}\ pm, stack \quad \text{(Tagged expr apply)}$$

Third, the roles of a CryptoVerif process cannot be executed in any order: if a role is defined after the return from an oracle, it can be executed only after the previous oracle has returned. For instance, we can run a server only after generating its keys. We need to enforce this constraint also in the OCaml program. Each CryptoVerif role role is translated by our compiler into an OCaml module μ_{role}. We add to the OCaml configuration the multiset of callable modules $\mathcal{MI}$ that contains pairs (μ_{role}, b) of a module μ_{role} and a boolean b, indicating, if true, that the module can be called any number of times and if false that the module can be called only once. Hence, the instrumented semantic configuration is

$$\mathcal{CI} = [Th_1, \ldots, Th_n], globalstore, j, \mathcal{MI}, events$$

We adapt the toplevel semantic rules to this configuration as shown in Figure 6.9. The semantic rules (New toplevel), (New toplevel schedule1), and (New toplevel schedule2) are straightforwardly adapted from the corresponding rules in the non-instrumented semantics by adding the components $\mathcal{MI}, events$. The rule (Toplevel event) gives the semantics of event: it adds its argument $e(v_1, \ldots, v_n)$ to the list *events* in the configuration and returns $(v_1, \ldots, v_n)$. The rule (New toplevel add thread) gives the instrumented semantics of addthread: the addthread construct is modified to reject new programs that contain a module that cannot be called. We let $\mathcal{M}_g$ be the set of generated modules. The programs spawned by addthread can be of two forms. Either they are *attacker programs* that contain neither the module corresponding to the primitives μ_{prim} nor any generated module in $\mathcal{M}_g$, or they are *protocol programs* that first contain the module corresponding to the primitives μ_{prim}, then the necessary generated modules $\mu_1, \ldots, \mu_l$ in $\mathcal{M}_g$, and finally any non-generated program *program′*. (We require this order on the modules for simplicity.) The generated modules $\mu_1, \ldots, \mu_l$ must be callable according to the value of $\mathcal{MI}$. The modules $\mu_1, \ldots, \mu_l$ that can be called only once are removed from the callable modules by removing the multiset $\mathcal{MI}'$ from $\mathcal{MI}$.

We also add the expression $\mathsf{return}(\mathcal{MI}', e)$ that adds to the multiset $\mathcal{MI}$ the generated modules present in $\mathcal{MI}'$, and returns the result of e, as defined by rule (Toplevel return). This expression is useful to add modules newly defined at the return from an oracle. We also add the minimal expression evaluation context $\mathsf{return}(\mathcal{MI}, [\cdot])$ to be able to evaluate the second argument of return.

Figure 6.9 Updated toplevel rules for the instrumented semantics

$$\frac{Th, globalstore \longrightarrow_p Th', globalstore'}{[Th_1, \ldots, Th_{tj-1}, Th, Th_{tj+1}, \ldots, Th_n], globalstore, tj, \mathcal{MI}, events \longrightarrow_p [Th_1, \ldots, Th_{tj-1}, Th', Th_{tj+1}, \ldots, Th_n], globalstore', tj, \mathcal{MI}, events}$$

(New toplevel)

$$\frac{\begin{array}{l} program = program(\mu_{\mathsf{prim}});; program(\mu_1);; \ldots;; program(\mu_l);; program' \\ program' \text{ does not contain } program(\mu_{\mathsf{prim}}) \text{ nor any } program(\mu) \text{ for } \mu \in \mathcal{M}_g \\ \mathcal{M} = \{\mu_1, \ldots, \mu_l\} \subseteq \mathcal{M}_g \\ \forall \mu \in \mathcal{M}, \exists b, (\mu, b) \in \mathcal{MI} \\ \mathcal{MI}' = \{(\mu, \text{false}) \mid \mu \in \mathcal{M} \wedge (\mu, \text{false}) \in \mathcal{MI}\} \\ \text{or} \\ program \text{ does not contain } program(\mu_{\mathsf{prim}}) \text{ nor any } program(\mu) \text{ for } \mu \in \mathcal{M}_g \\ \mathcal{M} = \emptyset, \mathcal{MI}' = \emptyset \end{array}}{\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{addthread}(program), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n, \langle \emptyset, program, [\,], \emptyset\rangle], \\ \quad globalstore, tj, \mathcal{MI} \setminus \mathcal{MI}', events \end{array}}$$

(New toplevel add thread)

$$\frac{1 \leq tj' \leq n}{\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{schedule}(tj'), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj', \mathcal{MI}, events \end{array}}$$

(New toplevel schedule1)

$$\frac{tj' < 1 \text{ or } tj' > n}{\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{schedule}(tj'), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{raise\ Invalid\_argument}, stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \end{array}}$$

(New toplevel schedule2)

$$\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{return}(\mathcal{MI}', v), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, v, stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI} \cup \mathcal{MI}', events \end{array}$$

(Toplevel return)

$$\begin{array}{l} [Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{event}\ ev(v_1, \ldots, v_n), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, events \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (v_1, \ldots, v_n), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{MI}, e(v_1, \ldots, v_n) :: events \end{array}$$

(Toplevel event)

Let us now show that this instrumentation does not alter the semantics of OCaml: an instrumented program behaves exactly in the same way as that program with the instrumentation deleted, provided only allowed roles are executed, as assumed by Assumption A2. This assumption is formalized as follows:

Assumption 6.1 (Only allowed roles) *The instrumented* addthread *rule* (New toplevel add thread) *never fails.*

We first show that, when a program or expression is a value v or an exceptional value raise v, the environment does not matter. To prove this property, we define the following equivalence.

Definition 6.2 *We define the equivalence* $\approx_{vTh}$ *on threads by*

$$\langle env, pe, stack, store\rangle \approx_{vTh} \langle env', pe', stack', store'\rangle$$

if and only if $pe, stack, store = pe', stack', store'$*, and if* pe *is not a value* v *or an exceptional value* raise v*, then* $env = env'$*.*

We extend this equivalence to non instrumented configurations $\mathcal{C}$ *and* $\mathcal{C}'$ *by* $\mathcal{C} \approx_v \mathcal{C}'$ *if and only if*

- $\mathcal{C} = [Th_1, \ldots, Th_n], globalstore, tj$,
- $\mathcal{C}' = [Th'_1, \ldots, Th'_n], globalstore, tj$,
- $\forall tj' \leq n, Th_{tj'} \approx_{vTh} Th'_{tj'}$.

We show that configurations equivalent by $\approx_v$ reduce in the same way.

Lemma 6.3 *If* $\mathcal{C} \approx_v \mathcal{C}'$ *and* $\mathcal{C} \rightarrow_p \mathcal{C}''$*, then* $\mathcal{C}' \rightarrow_p \mathcal{C}'''$ *and* $\mathcal{C}'' \approx_v \mathcal{C}'''$*.*

Proof No semantic rule uses the environment when the program or expression is a value or an exceptional value.

Indeed, the only semantic rules that apply when the program or expression is a value or an exceptional value are (Context out), (Context raise1), (Context raise2), (Let ctx out), and (Let ctx raise). All these rules replace the current environment with the one stored at the top of the stack. □

By reviewing the changes to the semantics, we can see that the total probability of all reductions is still 1 for the instrumented semantics: If an instrumented semantic configuration $\mathcal{CI}$ can reduce, then

$$\sum_{\{\mathcal{CI}' | \mathcal{CI} \rightarrow_{p(\mathcal{CI}')} \mathcal{CI}'\}} p(\mathcal{CI}') = 1 .$$

Moreover, for each reduction $\mathcal{CI} \rightarrow_p \mathcal{CI}'$, we have $p > 0$.

Let us now define the function $\mathsf{noinstr}_{\mathcal{CI}}$ that takes a configuration in the instrumented semantics and returns the corresponding configuration in the non-instrumented semantics.

Definition 6.4 *The function* $\mathsf{noinstr}_{Th1}$ *applied to a thread replaces*

1. *every* $\mathsf{return}(\mathcal{MI}, e)$ *with* e,
2. *every* event $ev(e_1, \ldots, e_n)$ *with* $(e_1, \ldots, e_n)$,
3. *and all* tagfunction *functions and closures with regular ones*

in this thread.

The function $\mathsf{noinstr}_{Th2}$ *modifies the stack of the thread by*

- *removing any pair of the form* $(env, \mathsf{return}(\mathcal{MI}, [\cdot]))$,
- *and transforming each pair of the form* $(env, \mathsf{event}(ev(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)))$ *into the pair* $(env, (e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n))$.

Let $\mathsf{noinstr}_{Th} \stackrel{\text{def}}{=} \mathsf{noinstr}_{Th1} \circ \mathsf{noinstr}_{Th2}$.
Finally, let us define

$$\mathsf{noinstr}_{\mathcal{CI}}([Th_1, \ldots, Th_n], globalstore, tj, \mathcal{MI}, events) \stackrel{\text{def}}{=} [\mathsf{noinstr}_{Th}(Th_1), \ldots, \mathsf{noinstr}_{Th}(Th_n)], globalstore, tj$$

We do not need to replace elements of the global store, as they cannot contain closures: event, return, and tagged functions cannot appear in them.

The next proposition shows that, with Assumption 6.1, there is a weak bisimulation between the non-instrumented semantics and the instrumented semantics, that is, the reductions match in the two semantics, but the number of steps may differ. Indeed, the return and event expressions introduce an additional transition in the instrumented semantics. All other constructs reduce in the same number of steps in both semantics. Hence, the instrumentation does not alter the semantics of the language.

Proposition 6.5 *1. If* $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ *and* $\mathcal{C}_1, \ldots, \mathcal{C}_n$ *are pairwise distinct configurations such that for all* $i \leq n$, *we have* $\mathcal{C} \rightarrow_{p_i} \mathcal{C}_i$ *with* $\sum_{i \leq n} p_i = 1$, *then there exist pairwise distinct instrumented configurations* $\mathcal{CI}_1, \ldots, \mathcal{CI}_n$ *such that for all* $i \leq n$, *we have* $\mathcal{CI} \rightarrow^*_{p_i} \mathcal{CI}_i$ *and* $\mathcal{C}_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_i)$.

2. If $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ *and* $\mathcal{CI}_1, \ldots, \mathcal{CI}_n$ *are pairwise distinct instrumented configurations such that for all* $i \leq n$, *we have* $\mathcal{CI} \rightarrow_{p_i} \mathcal{CI}_i$ *with* $\sum_{i \leq n} p_i = 1$, *then there exist pairwise distinct configurations* $\mathcal{C}_1, \ldots, \mathcal{C}_n$ *such that for all* $i \leq n$, *we have* $\mathcal{C} \rightarrow^*_{p_i} \mathcal{C}_i$ *and* $\mathcal{C}_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_i)$.

Proof Let $n_{\mathsf{ev,ret}}(\mathcal{CI})$ be the number of occurrences of event or return in the instrumented configuration $\mathcal{CI}$. Let us first prove the following property:

2'. If $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ and $\mathcal{CI}_1, \ldots, \mathcal{CI}_n$ are pairwise distinct instrumented configurations such that for all $i \leq n$, $\mathcal{CI} \rightarrow_{p_i} \mathcal{CI}_i$ with $\sum_{i \leq n} p_i = 1$, then one of the following two properties holds:

P1. $n = 1$, $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_1)$, and $n_{\mathsf{ev,ret}}(\mathcal{CI}_1) < n_{\mathsf{ev,ret}}(\mathcal{CI})$.

P2. there exist pairwise distinct configurations $\mathcal{C}_1, \ldots, \mathcal{C}_n$ such that for all $i \leq n$, we have $\mathcal{C} \rightarrow_{p_i} \mathcal{C}_i$ and $\mathcal{C}_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_i)$.

We prove this property by case analysis on the possible reductions of $\mathcal{CI}$.

Let us first suppose that $\mathcal{CI}$ reduces by (Globalstore1) and (Toplevel) from a reduction $Th \rightarrow_p Th'$ of the current thread, and let us distinguish cases depending on the latter reduction:

- The reduction comes from rules (Context in) or (Let ctx in): we have $Th = \langle env, C[e], stack, store\rangle \rightarrow Th' = \langle env, e, (env, C) :: stack, store\rangle$ where e is not a value and C is a minimal expression or program evaluation context. Let us distinguish cases on the form of C.
  - If $C = \mathsf{return}(\mathcal{MI}, [\cdot])$, then we have $\mathsf{noinstr}_{Th}(Th) = \mathsf{noinstr}_{Th}(\langle env, e, stack, store\rangle) = \mathsf{noinstr}_{Th}(Th')$ by Definition 6.4, so by expanding this property to the complete configuration, and noting that the reduction removes one return, Property P1 holds.
  - If $C = \mathsf{event}\ ev(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)$, then we have by Definition 6.4,

    $$\begin{aligned}&\mathsf{noinstr}_{Th}(Th) = \\ &\quad \mathsf{noinstr}_{Th}(\langle env, (e_1, \ldots, e_{i-1}, e, v_{i+1}, \ldots, v_n), stack, store\rangle)\,, \\ &\mathsf{noinstr}_{Th}(Th') = \mathsf{noinstr}_{Th}(\\ &\quad \langle env, e, (env, (e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)) :: stack, store\rangle)\,,\end{aligned}$$

    and we have $\mathsf{noinstr}_{Th}(Th) \rightarrow \mathsf{noinstr}_{Th}(Th')$, so Property P2 holds.
  - If C is neither a return nor an event context, then the reduction $Th \rightarrow_p Th'$ implies $\mathsf{noinstr}_{Th}(Th) \rightarrow_p \mathsf{noinstr}_{Th}(Th')$, so Property P2 holds.
- The reduction comes from rules (Context out) or (Let ctx out): we have $Th = \langle env, v, (env', C) :: stack, store\rangle \rightarrow Th' = \langle env', C[v], stack, store\rangle$ where C is a minimal expression or program evaluation context. Let us distinguish cases on the form of C.
  - If $C = \mathsf{return}(\mathcal{MI}, [\cdot])$, then we have by Definitions 6.4 and 6.2,

    $$\begin{aligned}\mathsf{noinstr}_{Th}(Th) &= \mathsf{noinstr}_{Th}(\langle env, v, stack, store\rangle) \\ &\approx_{v\,Th} \mathsf{noinstr}_{Th}(Th')\,,\end{aligned}$$

    so by expanding this property to the complete configuration, and noting that the reduction removes one return, Property P1 holds.
  - If $C = \mathsf{event}\ ev(e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)$, then we have by Definition 6.4,

    $$\begin{aligned}&\mathsf{noinstr}_{Th}(Th) = \mathsf{noinstr}_{Th}(\\ &\quad \langle env, v, (env', (e_1, \ldots, e_{i-1}, [\cdot], v_{i+1}, \ldots, v_n)) :: stack, store\rangle)\,, \\ &\mathsf{noinstr}_{Th}(Th') = \\ &\quad \mathsf{noinstr}_{Th}(\langle env', (e_1, \ldots, e_{i-1}, v, v_{i+1}, \ldots, v_n), stack, store\rangle)\,,\end{aligned}$$

    so Property P2 holds.

  - If C is neither a return nor an event context, then Property P2 holds.

- The cases of (Context raise2) and (Let ctx raise) are similar to the previous case: Property P1 holds when the context is $\mathsf{return}(\mathcal{MI}, [\cdot])$; Property P2 holds otherwise.

- Property P2 holds in the other cases.

If $\mathcal{CI} \to \mathcal{CI}_1$ by (Toplevel return), then the program of the current thread is $\mathsf{return}(\mathcal{MI}, v)$ in $\mathcal{CI}$ and the only differences between $\mathcal{CI}$ and $\mathcal{CI}_1$ are that the program of the current thread is v and the set of callable modules is changed in $\mathcal{CI}_1$. Therefore, $\mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}) = \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_1)$, and the reduction removes one return, so Property P1 holds.

If $\mathcal{CI} \to \mathcal{CI}_1$ by (Toplevel event), then the program of the current thread is $\mathsf{event}\ ev(v_1, \ldots, v_n)$ in $\mathcal{CI}$ and the only differences between $\mathcal{CI}$ and $\mathcal{CI}_1$ are that the program of the current thread is $(v_1, \ldots, v_n)$ and the list of executed events is updated in $\mathcal{CI}_1$. Therefore, $\mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}) = \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_1)$, and the reduction removes one event, so Property P1 holds.

Property P2 holds for $\mathsf{addthread}$, $\mathsf{schedule}$, and global store related reductions. In the case of $\mathsf{addthread}$, we use Assumption 6.1.

We have proved that Property 2' holds. Property 2 also holds, since it is a special case of Property 2'.

Let us now prove:

3. If $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ and $\mathcal{CI}$ cannot reduce, then $\mathcal{C}$ cannot reduce.

We need to prove that $\mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ cannot reduce. We can then use Lemma 6.3 to conclude. We distinguish cases depending on the program or expression in the current thread of $\mathcal{CI}$. If this program or expression was of the form $C[e]$ for some program or expression minimal evaluation context C, or $\mathsf{return}(\mathcal{MI}, v)$, or $\mathsf{event}\ ev(v_1, \ldots, v_n)$, the configuration $\mathcal{CI}$ could reduce. In all other cases, $\mathsf{noinstr}_{\mathcal{CI}}$ does not change the form of the possible reductions (since it transforms tagged functions into functions that behave exactly in the same way). Property 3 is true.

Let us now prove Property 1 by induction on $n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI})$. Let us suppose that $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI})$ and $\mathcal{C}_1, \ldots, \mathcal{C}_n$ are pairwise distinct configurations such that for all $i \leq n$, we have $\mathcal{C} \to_{p_i} \mathcal{C}_i$ with $\sum_{i \leq n} p_i = 1$.

The configuration $\mathcal{CI}$ must reduce, otherwise, by Property 3, the configuration $\mathcal{C}$ would also not reduce. Let $\mathcal{CI} \to_{p'_i} \mathcal{CI}_i$ for $i \leq n'$ be all the reductions possible from $\mathcal{CI}$. By Property 2', we are either in case P1 or in case P2.

In case P1, $\mathcal{CI}$ reduces into only one configuration $\mathcal{CI}_1$ such that $\mathcal{C} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_1)$, and $n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI}_1) < n_{\mathsf{ev},\mathsf{ret}}(\mathcal{CI})$. By induction hypothesis, there exist pairwise distinct instrumented configurations $\mathcal{CI}'_1, \ldots, \mathcal{CI}'_n$ such that for all $i \leq n$, we have $\mathcal{CI}_1 \to^*_{p_i} \mathcal{CI}'_i$ and $\mathcal{C}_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}'_i)$. As there is only one reduction from $\mathcal{CI}$ to $\mathcal{CI}_1$ with probability 1, we can conclude that Property 1 holds in this case.

In case P2, there exist pairwise distinct configurations $\mathcal{C}'_1, \ldots, \mathcal{C}'_{n'}$ such that for all $i \leq n'$, we have $\mathcal{C} \to_{p'_i} \mathcal{C}'_i$ and $\mathcal{C}'_i \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_i)$. Since $\sum_{i \leq n} p_i = 1$ and $\sum_{i \leq n'} p'_i = 1$, the reductions $\mathcal{C} \to_{p_i} \mathcal{C}_i$ $(i \leq n)$ are all possible reductions of $\mathcal{C}$

and the reductions $\mathcal{C} \rightarrow_{p'_i} \mathcal{C}'_i$ $(i \leq n')$ are also all possible reductions of $\mathcal{C}$, so they are the same reductions. Therefore, $n = n'$ and there exists a bijection α from $\{1, \ldots, n\}$ to $\{1, \ldots, n\}$ such that $p_i = p'_{\alpha(i)}$, $\mathcal{C}_i = \mathcal{C}'_{\alpha(i)} \approx_v \mathsf{noinstr}_{\mathcal{CI}}(\mathcal{CI}_{\alpha(i)})$, and $\mathcal{CI} \rightarrow_{p_i} \mathcal{CI}_{\alpha(i)}$. By renumbering the configurations $\mathcal{CI}_i$ $(i \leq n)$, we can conclude that Property 1 holds in this case. □

In the following chapters, we use only the instrumented semantics. Furthermore, we denote instrumented configurations by $\mathcal{C}$ to lighten notations.

Chapter 7

Translation Adaptation

The translation we described in Chapter 3 did not use the instrumentation we added to the OCaml language in Chapter 6. We need to modify some of the translation rules to use our instrumentation, and we also need to adapt the translation to our model of files and tables, which are contained in the global store. We finish by describing assumptions on our OCaml adversary $program_0$, and the initial OCaml configuration.

7.1 Changes Prompted by Our Instrumentation

Our OCaml configuration includes the multiset $\mathcal{MI}$, which contains the modules generated by our translation that can now to be run. We therefore need to update this multiset when new modules become available on oracle termination. In Chapter 3, we translate a role role by taking the process $Q(\mathsf{role})$ that is enclosed between the role declaration $\mathsf{role}\,\{$, and the closing braces $\}$. We must be aware of the role definitions that may come after this closing brace to be able to add them to $\mathcal{MI}$, so our translation function now takes the entire subprocess under the definition of the role.

As the processes we translate now may contain role declarations, we need to adapt the definition of the function reduce$'$, which returns the first oracles present in an oracle definition Q. We add the following rule:

$$\text{reduce}'(\mathsf{role}\,\{Q) \stackrel{\text{def}}{=} [\,] \qquad \text{(Role)}$$

The function reduce$'$ takes processes Q that follow return statements that do not end a role. By Assumption 5.3, we are inside a role, so by Property 1.6, the construct $\mathsf{role}\,\{Q'$ cannot appear in Q before a return statement that ends the current oracle. So, the function reduce$'$ will never be called on $\mathsf{role}\,\{Q'$: we chose to return an empty list in this case.

We also define the function $\mathbb{G}_{\mathsf{get}\mathcal{MI}}$ that returns a description of the modules that correspond to roles defined at the beginning of an oracle definition Q. The function $\mathbb{G}_{\mathsf{get}\mathcal{MI}}$ is similar to the function reduce$'$ above: it returns pairs containing the module generated for the role and a boolean indicating whether the role is under replication or not. In contrast to reduce$'$, it returns a set and not a list.

$$\mathbb{G}_{\mathsf{get}\mathcal{MI}}(0) \stackrel{\text{def}}{=} \emptyset \qquad \text{(Nil)}$$

$$\mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q_1 \mid Q_2) \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q_1) \cup \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q_2) \quad \text{(Par)}$$

$$\mathbb{G}_{\mathsf{get}\mathcal{MI}}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} \{(\mu, \text{true}) \mid \exists b, (\mu, b) \in \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q)\} \quad \text{(Repl)}$$

$$\mathbb{G}_{\mathsf{get}\mathcal{MI}}(O[\widetilde{i}](x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]) := P) \stackrel{\text{def}}{=} \emptyset \quad \text{(Oracle)}$$

$$\mathbb{G}_{\mathsf{get}\mathcal{MI}}(\mathsf{role}\,\{Q) \stackrel{\text{def}}{=} \{(\mu_{\mathsf{role}}, \text{false})\} \quad \text{(Role)}$$

The function $\mathbb{G}_{\mathsf{get}\mathcal{MI}}$ takes processes Q that follow return statements that end the current role. By Assumption 5.3, there cannot be an oracle definition before a $\mathsf{role}\,\{Q'$ in Q. So the function $\mathbb{G}_{\mathsf{get}\mathcal{MI}}$ will never be called on oracle definitions.

We also need to modify the translation of the return construct to distinguish whether the current role finishes or not. We change the translation rule (Return) into:

$$\frac{[(Q_1, b_1), \ldots, (Q_l, b_l)] \stackrel{\text{def}}{=} \text{reduce}'(Q)}{\mathbb{G}(\mathsf{return}(N_1, \ldots, N_k); Q) \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_l, b_l), \mathbb{G}_{\mathsf{M}}(N_1), \ldots, \mathbb{G}_{\mathsf{M}}(N_k))} \quad \text{(Return1)}$$

$$\mathbb{G}(\mathsf{return}(N_1, \ldots, N_k)\}; Q) \stackrel{\text{def}}{=} (\mathsf{return}(\mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q), (\mathbb{G}_{\mathsf{M}}(N_1), \ldots, \mathbb{G}_{\mathsf{M}}(N_k)))) \quad \text{(Return2)}$$

We translate a return that finishes the current role to the term that adds the roles declared in Q to the multiset $\mathcal{MI}$, and returns the translation of the terms $N_1, \ldots, N_k$.

We have defined in our instrumentation the expression $\mathsf{event}\ ev(v_1, \ldots, v_k)$ that, similarly to CryptoVerif events, adds the event $ev(v_1, \ldots, v_k)$ to the list of events *events* and returns the tuple $(v_1, \ldots, v_k)$. We modify the rule (Event) in order to have a correspondence between OCaml and CryptoVerif events.

$$\mathbb{G}(\mathsf{event}\ ev(M_1, \ldots, M_k); P) \stackrel{\text{def}}{=} \mathsf{event}\ ev(\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_k)); \mathbb{G}(P) \quad \text{(Event)}$$

Tagged functions have been added to the language to be able to distinguish between attacker-created functions and closures on the one hand and our generated functions and closures on the other hand. We modify the translation of an oracle and the initialization function of the module to use these tagged closures:

$$\mathbb{G}_{\mathsf{O}}(Q, \text{false}) \stackrel{\text{def}}{=} \mathsf{let}\ token = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}^{O}\ pm_{\text{false}}(Q) \quad \text{(Oracle1)}$$

$$\mathbb{G}_{\mathsf{O}}(Q, \text{true}) \stackrel{\text{def}}{=} \mathsf{tagfunction}^{O}\ pm_{\text{true}}(Q) \quad \text{(Oracle2)}$$

$$program(\mu_{\mathsf{role}}) \stackrel{\text{def}}{=} \mathsf{let}\ \mu_{\mathsf{role}}.\mathsf{init} = \mathsf{let}\ token = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}}\ pm_{\mathsf{role}} \quad \text{(Init)}$$

7.2 Changes Prompted by Our Model

As explained in Chapter 6, files are modeled as a location in the global store. Therefore, we change the definition of $\mathbb{G}_{\mathsf{file}}$: $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]) \stackrel{\text{def}}{=} (f := \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x))$ if $(x[\widetilde{i}], f) \in \mathsf{files}$ and $\mathbb{G}_{\mathsf{file}}(x[\widetilde{i}]) \stackrel{\text{def}}{=} ()$ otherwise.

We represent table contents by lists of tuples of bitstrings, and we replace calls to read_table and add_to_table in the translation with the implementation in

our model. We inline these calls in order to simplify the proof of correctness of the translation: entering and quitting functions involve context changes in the semantics, which complicates slightly the reductions.

$$\frac{(Tbl, f) \in \mathsf{tables}}{\begin{array}{l}\mathbb{G}(\mathsf{insert}\ Tbl(M_1, \ldots, M_k); P) \stackrel{\text{def}}{=} \\ \quad (f := (\mathbb{G}_{\mathsf{ser}}(T_{M_1})\ \mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{ser}}(T_{M_k})\ \mathbb{G}_{\mathsf{M}}(M_k)) :: (!f); \mathbb{G}(P))\end{array}} \quad \text{(Insert)}$$

$$\mathbb{G}_{\mathsf{fold}} \stackrel{\text{def}}{=} f \to \mathsf{function}\ a \to \mathsf{function}\ [\,] \to a \mid x :: l \to f\ (\mathit{fold}\ f\ a\ l)\ x \quad \text{(Fold)}$$

$$\begin{array}{l}\mathbb{G}_{\mathsf{read\_table}}(f, c) \stackrel{\text{def}}{=} \mathsf{let\ rec}\ \mathit{fold} = \mathsf{function}\ \mathbb{G}_{\mathsf{fold}}\ \mathsf{in} \\ \quad \mathit{fold}\ (\mathsf{function}\ a \to \mathsf{function}\ x \to \\ \qquad (\mathsf{try}\ (c\ x) :: a\ \mathsf{with} \\ \qquad\quad \mathsf{Match\_failure} \to a))\ [\,]\ !f\end{array} \quad \text{(Read table)}$$

$$\frac{(Tbl, f) \in \mathsf{tables}}{\begin{array}{l}\mathbb{G}(\mathsf{get}\ Tbl(x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=} \\ \quad \mathsf{let}\ l = \mathbb{G}_{\mathsf{read\_table}}(f, \mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M))\ \mathsf{in} \\ \quad \mathsf{if}\ l = [\,]\ \mathsf{then}\ \mathbb{G}(P') \\ \qquad\quad \mathsf{else\ let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = \mathsf{random}_l\ l\ \mathsf{in} \\ \qquad\quad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P))\end{array}} \quad \text{(Get)}$$

The new (Insert) translation rule first gets the contents of the file containing the contents of the table Tbl by dereferencing the global store location f, then adds the translation of the term $M_1, \ldots, M_k$ to $!f$, and finally stores this new list back into f.

The expression $\mathbb{G}_{\mathsf{read\_table}}$ has the same behavior as the function read_table: it takes as arguments a global store location f that corresponds to a table, and a closure filter, and returns the list of values filter e for all elements e in $!f$ such that filter e does not raise the exception raise Match_failure. The translation of get is modified to use $\mathbb{G}_{\mathsf{read\_table}}$ instead of read_table.

7.3 The Adversary

The generated modules $\mathcal{M}_g$ (μ_{role} for each role in the CryptoVerif process) are included in manually-written programs that represent the full implementation of the protocol, for instance a client and a server. In particular, these programs are responsible for sending the result of oracles to the network and receiving messages to be passed as arguments to oracles. These programs interact with an adversary that we model as an OCaml program $\mathit{program}_0$. We consider that the programs of the protocol are launched by the adversary $\mathit{program}_0$ using the addthread construct. The generated modules depend only on the module containing the cryptographic primitives μ_{prim}, so when the program of a thread uses the primitives or the generated modules, we can order the programs of the modules in the argument of addthread in the order $\mathit{program}(\mu_{\mathsf{prim}});; \mathit{program}(\mu_{\mathsf{role}_1});; \ldots;; \mathit{program}(\mu_{\mathsf{role}_k});; \mathit{program}'$ where $\mathit{program}'$ contains no generated module, as required by the instrumented semantics of addthread (New toplevel add thread).

We assume that $program_0$ uses the generated modules only inside addthread, and that $program_0$ is a well-typed OCaml program. (The network code is well-typed by Assumption A4. The adversary itself is any probabilistic Turing machine, which can be implemented by a well-typed OCaml program.) We assume that only the generated modules use events, tagged functions, and return. The adversary must not use events, which serve for specifying security properties of the protocol, nor return, which serves for updating the set of callable generated modules. He uses regular functions rather than tagged functions. Moreover, as mentioned in Assumption A3, we suppose that only the generated modules access files that contain private CryptoVerif data (free variables of roles and tables). So we let $S_{priv} \stackrel{\text{def}}{=} \{f \mid (x[\,], f) \in \mathsf{files} \text{ or } (Tbl, f) \in \mathsf{tables}\} \subseteq S_g$ be the set of global locations reserved for private CryptoVerif data, and we have the following assumption:

Assumption 7.1 *The locations in S_{priv} occur only in the programs of generated modules; they do not occur elsewhere in $program_0$.*

The program $program_0$ is run in the initial (instrumented) OCaml configuration $\mathcal{C}_0(Q_0, program_0)$ defined as follows:

$$\mathcal{C}_0(Q_0, program_0) \stackrel{\text{def}}{=} [\langle \emptyset, program_0, [\,], \emptyset \rangle], globalstore_0, 1, \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q_0), [\,]$$

where $\mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q_0)$ is the set of modules available at the beginning of the execution and $globalstore_0 \stackrel{\text{def}}{=} \{l \mapsto initval_l \mid l \in S_g\}$ is the initial value of global store, as defined in Section 6.1.6. Tables are represented by lists, and their initial value $initval_l$ is the empty list $[\,]$, representing that the tables are initially empty. Files that contain free variables of roles are represented by strings, and their initial value $initval_l$ is the empty string "". For other elements, the initial value $initval_l$ is the default value for the type of location l.

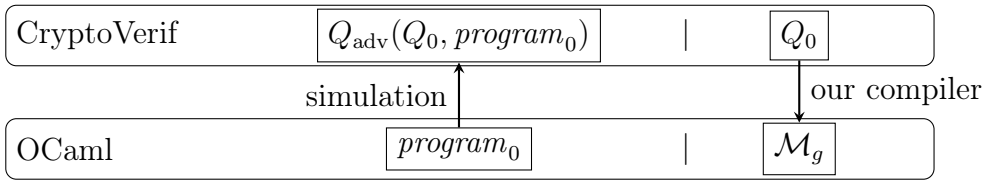
Chapter 8

Proof of Correctness

This section presents the proof of correctness of our compiler. We give ourselves a CryptoVerif process Q_0 that corresponds to a cryptographic protocol. By using our compiler, we first generate modules $\mathcal{M}_g$ that correspond to the roles present inside Q_0, as explained in the previous section. We consider an adversary interacting with the protocol implementation, modeled as an OCaml program $program_0$ that uses the generated modules in $\mathcal{M}_g$. Informally, when CryptoVerif shows that Q_0 satisfies a certain security property, it shows that for any CryptoVerif adversary Q, the probability that $Q \mid Q_0$ breaks the security property is bounded by a certain bound, which CryptoVerif computes. Our goal is to show that the same probability bound also applies to the generated implementation, that is, the probability that $program_0$ breaks the security property is bounded by the same bound. As illustrated in Figure 8.1, to prove this property, we build from the OCaml adversary $program_0$ a CryptoVerif adversary $Q_{\text{adv}}(Q_0, program_0)$ that simulates $program_0$. Basically, we run the OCaml program $program_0$ inside a CryptoVerif primitive *simulate*; when $program_0$ would call the translation of an oracle in $\mathcal{M}_g$, $Q_{\text{adv}}(Q_0, program_0)$ calls the corresponding oracle in Q_0. We prove that $Q_{\text{adv}}(Q_0, program_0) \mid Q_0$ and $program_0$ using $\mathcal{M}_g$ behave similarly, hence they have the same probability of breaking the security property. To achieve this goal, we need to prove, firstly, that the translations of the oracles behave in the same way as the CryptoVerif oracles, and secondly, that our simulation is sound.

In Section 8.1, we state our assumptions on the cryptographic primitives, and show that the primitives behave correctly independently of the rest of the program. In Section 8.2, we prove that the OCaml translation of a CryptoVerif oracle behaves like the oracle. In Section 8.3, we define the CryptoVerif adversary that simulates the OCaml adversary $program_0$. Finally, in Section 8.4, we prove

Figure 8.1 Overview of our proof



that the CryptoVerif adversary interacting with Q_0 behaves like the OCaml adversary interacting with the generated implementation. This result shows the desired correctness of our compiler.

8.1 Correctness of Cryptographic Primitives

Let us first formalize the assumptions we make about the implementation of cryptographic primitives. Let $program_{\mathsf{prim}} \stackrel{\text{def}}{=} program(\mu_{\mathsf{prim}})$ be the program of the module that defines the primitives and $interface_{\mathsf{prim}} \stackrel{\text{def}}{=} interface(\mu_{\mathsf{prim}})$ be its interface. The interface $interface_{\mathsf{prim}}$ consists of the function random_l, the functions $\mathbb{G}_{\mathsf{f}}(f)$ for each CryptoVerif function f, and the functions $\mathbb{G}_{\mathsf{random}}(T)$, $\mathbb{G}_{\mathsf{ser}}(T)$, $\mathbb{G}_{\mathsf{deser}}(T)$, and $\mathbb{G}_{\mathsf{pred}}(T)$ for each CryptoVerif type T for which these functions are used in the translation, as described in Section 1.3. (The functions $\mathbb{G}_{\mathsf{ser}}(T)$ and $\mathbb{G}_{\mathsf{deser}}(T)$ are either both present or both absent in $interface_{\mathsf{prim}}$.) We rely on the following assumptions.

Assumption 8.1 *There are no* schedule, addthread, return, *nor* event *operations and no global store locations in* $program_{\mathsf{prim}}$.

An OCaml semantic configuration in which the current thread does not use addthread, return, event, schedule operations, nor global store locations reduces by using the rule (Thread) of Figure 6.7, so we can reduce it by considering as configuration only a thread Th. We denote by $\mathcal{TT}$ the traces over threads.

Let $Th_0^s \stackrel{\text{def}}{=} \langle \emptyset, program_{\mathsf{prim}};;, [\,], \emptyset \rangle$ be a thread configuration that evaluates only the implementation of the cryptographic primitives module.

Assumption 8.2 *There exists a unique complete thread trace $\mathcal{TT}$ beginning at the configuration Th_0^s and there exists env_{prim} such that the last configuration of the trace $\mathcal{TT}$ is:*

$$Th = \langle env_{\mathsf{prim}}, \varepsilon, [\,], \emptyset \rangle$$

This assumption means that there are no uncatched exceptions, no access to the store, and no random operations in the initialization of the module μ_{prim}, so that the environment env_{prim} is always the same. Typically, the initialization just defines functions, so this assumption is not restrictive. Random choices and a limited access to the store explained below are allowed during calls to primitives. By definition of a module, we have $interface_{\mathsf{prim}} \subseteq \mathrm{Dom}(env_{\mathsf{prim}})$.

Assumption 8.3 *For each CryptoVerif type T, OCaml values of the corresponding type $\mathbb{G}_T(T)$ does not contain closures nor store or global store locations.*

In particular, bitstrings received or returned by cryptographic primitives are typically represented by strings. Strings are values in our semantics, and do not contain locations. So one cannot alter the contents of a string after creation: string values are not mutable. In OCaml, the type `string` is mutable, so either one should use an immutable abstract type instead of `string` to represent bitstrings, or one needs to assume that the network code does not modify the

contents of strings that go through our generated closures, as mentioned in Assumption A5.

To establish the correspondence between CryptoVerif values and OCaml values, we define a function $\mathbb{G}_{\mathsf{val}T}$, which maps each CryptoVerif bitstring a to its associated value v in OCaml. For a given type T, $\mathbb{G}_{\mathsf{val}T}$ must be a bijection between T and the set of OCaml values of type $\mathbb{G}_T(T)$ satisfying the predicate function $\mathbb{G}_{\mathsf{pred}}(T)$. Furthermore, the OCaml value true and the CryptoVerif value true are such that $\mathbb{G}_{\mathsf{val}bool}(\text{true}) = \text{true}$, and the same goes for false. We extend this function to events by $\mathbb{G}_{\mathsf{ev}}(e(a_1, \ldots, a_j)) = e(\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_j}(a_j))$ if e is of type $T_1 \times \cdots \times T_j$. This function is naturally extended to lists of events.

The next assumption states that the primitives have been correctly implemented, following Assumption A1: the implementation of the cryptographic primitives in $interface_{\mathsf{prim}}$ correctly emulates the corresponding behavior of CryptoVerif.

Assumption 8.4 (Correct primitives) *For each CryptoVerif function f of type $T_1 \times \cdots \times T_n \to T$, for each CryptoVerif values $a_1, \ldots, a_n$ of types $T_1, \ldots, T_n$, there exist env and store such that*

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{f}}(f))\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_n}(a_n)), [\,], \emptyset\rangle \to^* \langle env, \mathbb{G}_{\mathsf{val}T}(f(a_1, \ldots, a_n)), [\,], store\rangle\,.$$

For each CryptoVerif type T such that the function $\mathbb{G}_{\mathsf{random}}(T)$ is in $interface_{\mathsf{prim}}$, for each CryptoVerif value $a \in T$, there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{random}}(T))\ (), [\,], \emptyset\rangle \to^*_{1/|T|} \langle env, \mathbb{G}_{\mathsf{val}T}(a), [\,], store\rangle\,.$$

For each CryptoVerif type T such that the function $\mathbb{G}_{\mathsf{pred}}(T)$ is in $interface_{\mathsf{prim}}$, for each value v of the OCaml type $\mathbb{G}_T(T)$, there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{pred}}(T))\ v, [\,], \emptyset\rangle \to^* \langle env, v', [\,], store\rangle$$

where $v' = \text{true}$ when $\mathbb{G}^{-1}_{\mathsf{val}T}(v)$ exists, and $v' = \text{false}$ otherwise.

For each CryptoVerif type T such that the functions $\mathbb{G}_{\mathsf{ser}}(T)$ and $\mathbb{G}_{\mathsf{deser}}(T)$ are in $interface_{\mathsf{prim}}$, for each CryptoVerif value $a \in T$, there exists an OCaml string value $\mathsf{ser}(T, a)$, such that there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{ser}}(T))\ \mathbb{G}_{\mathsf{val}T}(a), [\,], \emptyset\rangle \to^* \langle env, \mathsf{ser}(T, a), [\,], store\rangle$$

and there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{deser}}(T))\ \mathsf{ser}(T, a), [\,], \emptyset\rangle \to^* \langle env, \mathbb{G}_{\mathsf{val}T}(a), [\,], store\rangle\,.$$

If v is a non-empty list, then for each $a \in v$, there exist env and store such that

$$\langle \emptyset, env_{\mathsf{prim}}(\mathsf{random}_l)\ v, [\,], \emptyset\rangle \to^*_{\sum_{j \in S} \text{among}(\{1,\ldots,|v|\},j)} \langle env, a, [\,], store\rangle$$

where $S \stackrel{\text{def}}{=} \{1 \leq j \leq |v| \mid \mathsf{nth}(v, j) = a\}$.

In contrast to [21], we allow the cryptographic primitives to use the store for their internal computations (which often happens in practice). However, the primitives are not allowed to communicate across calls or to communicate data to the adversary or to the rest of the code using the store. They can create new locations and use these locations internally; these locations become unreachable as soon as the call to the primitive ends. This assumption corresponds to the intuition that the cryptographic primitives are pure functions: their usage of the store should not have any visible side effect. This assumption is modeled above by considering that the primitives are initially called in an empty store. Since their return value does not contain locations, the store at the end of the call will be unreachable.

The last statement of Assumption 8.4 guarantees that random_l is programmed correctly: $\mathsf{random}_l\ v$ returns a random element of the list v, such that the probability of returning the j-th element of v is among$(\{1, \ldots, |v|\}, j)$. In case the same element occurs several times in v, the probability of that element is then the sum of the probabilities of all its occurrences.

In general, when primitives make probabilistic choices, they might return the same result in several traces with a different environment and store. To simplify notations, Assumption 8.4 states that this does not happen, so that we have the same environment and store in all final configurations that yield the same result. Our proof could easily be extended to the general case if desired.

The next proposition shows that the primitives always return correct results, when they are called inside an OCaml program, so possibly with a non-empty store and a non-empty stack. It is a consequence of Assumption 8.4.

Proposition 8.5 (Correct behavior of the primitives) *Let us consider a thread* $Th \stackrel{\text{def}}{=} \langle env, env_{\mathsf{prim}}(s)\ v, stack, store\rangle$.

- *If* $s = \mathbb{G}_{\mathsf{f}}(f)$, f *is a CryptoVerif function of type* $T_1 \times \cdots \times T_n \to T$, *and* $v = (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_n}(a_n))$ *for some CryptoVerif values* $a_1, \ldots, a_n$ *of types* $T_1, \ldots, T_n$, *then there exist* env' *and* $store'$ *such that*

$$Th \to^* \langle env', \mathbb{G}_{\mathsf{val}T}(f(a_1, \ldots, a_n)), stack, store'\rangle\,.$$

- *If* $s = \mathbb{G}_{\mathsf{random}}(T)$ *and* $v = ()$, *then for each CryptoVerif value* $a \in T$, *there exist* env' *and* $store'$ *such that*

$$Th \to^*_{1/|T|} \langle env', \mathbb{G}_{\mathsf{val}T}(a), stack, store'\rangle\,.$$

- *If* $s = \mathbb{G}_{\mathsf{pred}}(T)$, *then there exist* env' *and* $store'$ *such that*

$$Th \to^* \langle env', v', stack, store'\rangle$$

where $v' = \text{true}$ *when* $\mathbb{G}^{-1}_{\mathsf{val}T}(v)$ *exists, and* $v' = \text{false}$ *otherwise.*

- *If* $s = \mathbb{G}_{\mathsf{ser}}(T)$ *and* $v = \mathbb{G}_{\mathsf{val}T}(a)$, *then there exist* env' *and* $store'$ *such that*

$$Th \to^* \langle env', \mathsf{ser}(T, a), stack, store'\rangle\,.$$

- *If* $s = \mathbb{G}_{\mathsf{deser}}(T)$ *and* $v = \mathsf{ser}(T, a)$*, then there exist* env' *and* $store'$ *such that*
$$Th \rightarrow^* \langle env', \mathbb{G}_{\mathsf{val}T}(a), stack, store' \rangle .$$
- *If* $s = \mathsf{random}_l$ *and* v *is a non-empty list, then for each* $a \in v$*, there exist* env' *and* $store'$ *such that*
$$Th \rightarrow^*_{\sum_{j \in S} \mathrm{among}(\{1,\ldots,|v|\},j)} \langle env', a, stack, store' \rangle$$
where $S \stackrel{\text{def}}{=} \{1 \leq j \leq |v| \mid \mathsf{nth}(v, j) = a\}$.

In all cases, we have $store' \supseteq store$.

To prove this proposition, we first prove a more general lemma, that proves that when we "plug" a thread in another, if the thread can reduce on its own, the plugged thread will reduce in a similar manner.

Definition 8.6 *Let* $Th \stackrel{\text{def}}{=} \langle env, pe, stack, store \rangle$ *and* $Th' \stackrel{\text{def}}{=} \langle env', pe', stack', store' \rangle$ *be two threads, such that the domains of store and store′ are disjoint.*

We define $plug(Th, Th') \stackrel{\text{def}}{=} \langle env, pe, stack \mathbin{@} stack', store \cup store' \rangle$.

Plugging a thread into another changes the thread by replacing the environment and expression by the first thread's environment and expression, and adds the stack and store to the stack and store present in the second thread.

Definition 8.7 *A well-formed thread* $Th = \langle env, pe, stack, store \rangle$ *is a thread such that:*

1. *all store locations* $l \in S_l$ *that occur in the thread Th are bound in the store:* $l \in \mathrm{Dom}(store)$,
2. *pe and stack do not contain global store locations, nor* return, event, schedule, *or* addthread *operations.*

Lemma 8.8 *If Th is a well-formed thread and* $Th \rightarrow_p Th'$*, then for all* Th'' *such that the domains of the stores of* Th'' *and of Th are disjoint, after renaming the fresh locations introduced in* $Th \rightarrow_p Th'$ *so that they do not occur in* Th''*, we have* $plug(Th, Th'') \rightarrow_p plug(Th', Th'')$ *and the domains of the stores of* Th'' *and of* Th' *are disjoint.*

Proof By reviewing the reduction rules, we have Property (P1): if $env, pe, stack \xrightarrow{L}_p env', pe', stack'$, then for every stack $stack''$, we have $env, pe, stack \mathbin{@} stack'' \xrightarrow{L}_p env', pe', stack' \mathbin{@} stack''$.

Let $Th \stackrel{\text{def}}{=} \langle env, pe, stack, store \rangle$ and $Th' \stackrel{\text{def}}{=} \langle env', pe', stack', store' \rangle$. Let us prove that, if $Th \rightarrow_p Th'$, then for every $Th'' \stackrel{\text{def}}{=} \langle env'', pe'', stack'', store'' \rangle$ such that $\mathrm{Dom}(store) \cap \mathrm{Dom}(store'') = \emptyset$, we have the reduction $plug(Th, Th'') = \langle env, pe, stack \mathbin{@} stack'', store \cup store'' \rangle \rightarrow_p plug(Th', Th'') = \langle env', pe', stack' \mathbin{@} stack'', store' \cup store'' \rangle$ with $\mathrm{Dom}(store') \cap \mathrm{Dom}(store'') = \emptyset$. We distinguish cases on the label L present in rule (Thread).

- if L is empty, then by (Store empty), $store = store'$. We conclude by Property (P1) and rule (Thread).
- if L is $!l = v$, by (Store lookup), the location l is in the domain of the store $store$, and $store(l) = v$, and $store = store'$. We also have $(store \cup store'')(l) = v$, so $store \cup store'' \xrightarrow{!l=v} store' \cup store''$. We conclude by Property (P1) and rule (Thread).
- if L is $l := v$, then by (Store assign), the location l is in the domain of the store $store$, and $store' = store[l \mapsto v]$. The domain of the store $store \cup store''$ also contains l, so $store \cup store'' \xrightarrow{l:=v} store' \cup store''$. We conclude by Property (P1) and rule (Thread).
- if L is $\mathsf{ref}\ v = l$, then by (Store alloc), $l \notin \mathrm{Dom}(store)$. By reviewing the uses of L of the form $\mathsf{ref}\ v = l$ in the reduction rules, we can deduce that $pe = \mathsf{ref}\ v$ and $pe' = l$. By Property 1 of Definition 8.7, the location l does not occur in Th. Let us take a location $l' \in S_l$ that is not in $\mathrm{Dom}(store) \cup \mathrm{Dom}(store'')$; we rename l into l'. As $l' \notin \mathrm{Dom}(store)$, the thread Th' becomes $\langle env', l', stack', store[l' \mapsto v] \rangle$, which is in the same equivalence class as Th', so we still designate this thread by Th'. Let $L' \stackrel{\text{def}}{=} (\mathsf{ref}\ v = l')$. By Property (P1), $env, pe, stack \mathbin{@} stack'' \xrightarrow{L'} env', l', stack' \mathbin{@} stack''$ and, since $l' \notin \mathrm{Dom}(store) \cup \mathrm{Dom}(store'')$, we have $store \cup store'' \xrightarrow{L'} store[l' \mapsto v] \cup store''$. We conclude that $plug(Th, Th'') = \langle env, pe, stack \mathbin{@} stack'', store \cup store'' \rangle \to plug(Th', Th'') = \langle env', l', stack' \mathbin{@} stack'', store[l' \mapsto v] \cup store'' \rangle$ by rule (Thread). □

Lemma 8.9 *Let Th be a well-formed thread. If $Th \to_p Th'$, then Th' is also well-formed.*

Proof Let us prove that both properties of Definition 8.7 are preserved.

- The only rule that may add new locations in the thread is (Store alloc), which creates a new location l and also adds it in the domain of the store. So Property 1 is preserved for Th'.
- By looking at the reduction rules, we can see that no rule can create global store locations or return, event, schedule, or addthread operations. So Property 2 is preserved for Th'.

Therefore, the thread Th' is also well-formed. □

Lemma 8.10 *If v is a value of the type of the argument of the primitive s, then the thread $\langle \emptyset, env_{\mathsf{prim}}(s)\ v, [\,], \emptyset \rangle$ is well-formed.*

Proof The thread $Th_0^s = \langle \emptyset, program_{\mathsf{prim}};;, [\,], \emptyset \rangle$ is well-formed, since it contains no locations in S_l and by Assumption 8.1, it contains no return, event, schedule, or addthread operations and no global store locations.

By Assumption 8.2, $Th_0^s \to^* Th = \langle env_{\mathsf{prim}}, \varepsilon, [\,], \emptyset \rangle$, so by Lemma 8.9, Th is also well-formed. Therefore, the thread $\langle \emptyset, env_{\mathsf{prim}}(s)\ v, [\,], \emptyset \rangle$ is well-formed,

since by Assumption 8.3, v contains no locations and no return, event, schedule, or addthread operations since it contains no closure, and env_{prim} contains no locations and no return, event, schedule, or addthread operations since Th is well-formed. □

Proof (of Proposition 8.5) By Assumption 8.4, we have reductions of the form

$$Th_1 \stackrel{\text{def}}{=} \langle \emptyset, env_{\mathsf{prim}}(s)\ v, [\,], \emptyset \rangle \rightarrow_p^* Th_1' \stackrel{\text{def}}{=} \langle env', v', [\,], store_1' \rangle\,.$$

Let $Th_2 \stackrel{\text{def}}{=} \langle env, (), stack, store \rangle$. By Lemma 8.10, Th_1 is well-formed, so by Lemmas 8.8 and 8.9,

$$Th = plug(Th_1, Th_2) \rightarrow_p^* plug(Th_1', Th_2) = \langle env', v', stack, store_1' \cup store \rangle\,.$$

Letting $store' \stackrel{\text{def}}{=} store_1' \cup store$, we obtain exactly the desired reductions $Th \rightarrow_p^* \langle env', v', stack, store' \rangle$, and $store' \supseteq store$. □

8.2 Correctness of the Translation of Oracle Bodies

In this section, we show the correctness of the translation of oracle bodies in our compiler: we show a correspondence between the semantics of the oracle body in CryptoVerif and the semantics of its translation into OCaml.

Let $\mathsf{fv}(M)$, $\mathsf{fv}(P)$, $\mathsf{fv}(Q)$ be the free variables of the CryptoVerif term M and processes P and Q, respectively, defined as follows:

$\mathsf{fv}(x[\widetilde{i}]) \stackrel{\text{def}}{=} \{x[\widetilde{i}]\}$
$\mathsf{fv}(f(M_1, \ldots, M_k)) \stackrel{\text{def}}{=} \bigcup_{i=1}^k \mathsf{fv}(M_i)$
$\mathsf{fv}(0) \stackrel{\text{def}}{=} \emptyset$
$\mathsf{fv}(Q \mid Q') \stackrel{\text{def}}{=} \mathsf{fv}(Q) \cup \mathsf{fv}(Q')$
$\mathsf{fv}(\mathsf{foreach}\ i \leq n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} \mathsf{fv}(Q)$
$\mathsf{fv}(O[\widetilde{i}](x_1[\widetilde{i}] : T_1, \ldots, x_k[\widetilde{i}] : T_k) := P) \stackrel{\text{def}}{=} \mathsf{fv}(P) \setminus \{x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]\}$
$\mathsf{fv}(\mathsf{return}(M_1, \ldots, M_k); Q) \stackrel{\text{def}}{=} \bigcup_{i=1}^k \mathsf{fv}(M_i) \cup \mathsf{fv}(Q)$
$\mathsf{fv}(\mathsf{end}) \stackrel{\text{def}}{=} \emptyset$
$\mathsf{fv}(x[\widetilde{i}] \stackrel{R}{\leftarrow} T; P) \stackrel{\text{def}}{=} \mathsf{fv}(P) \setminus \{x[\widetilde{i}]\}$
$\mathsf{fv}(x[\widetilde{i}] \leftarrow M; P) \stackrel{\text{def}}{=} \mathsf{fv}(P) \setminus \{x[\widetilde{i}]\}$
$\mathsf{fv}(\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=} \mathsf{fv}(M) \cup \mathsf{fv}(P) \cup \mathsf{fv}(P')$
$\mathsf{fv}(\mathsf{insert}\ Tbl(M_1, \ldots, M_k); P) \stackrel{\text{def}}{=} \bigcup_{i=1}^k \mathsf{fv}(M_i) \cup \mathsf{fv}(P)$
$\mathsf{fv}(\mathsf{get}\ Tbl(x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=}$
$\quad (\mathsf{fv}(M) \cup \mathsf{fv}(P) \setminus \{x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]\}) \cup \mathsf{fv}(P')$
$\mathsf{fv}(\mathsf{event}\ e(M_1, \ldots, M_k); P) \stackrel{\text{def}}{=} \bigcup_{i=1}^k \mathsf{fv}(M_i) \cup \mathsf{fv}(P)$
$\mathsf{fv}(\mathsf{let}\ (x_1[\widetilde{i}] : T_1, \ldots, x_{k'}[\widetilde{i}] : T_{k'}) = O[M_1, \ldots, M_l](M_1', \ldots, M_k')\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=}$
$\quad \bigcup_{i=1}^l \mathsf{fv}(M_i) \cup \bigcup_{i=1}^k \mathsf{fv}(M_i') \cup (\mathsf{fv}(P) \setminus \{x_1[\widetilde{i}], \ldots, x_{k'}[\widetilde{i}]\}) \cup \mathsf{fv}(P')$
$\mathsf{fv}(\mathsf{let}\ x[\widetilde{i}] : T = \mathsf{loop}\ O[M_1, \ldots, M_l](M')\ \mathsf{in}\ P\ \mathsf{else}\ P') \stackrel{\text{def}}{=}$
$\quad \bigcup_{i=1}^l \mathsf{fv}(M_i) \cup \mathsf{fv}(M') \cup (\mathsf{fv}(P) \setminus \{x[\widetilde{i}]\}) \cup \mathsf{fv}(P')$

The only unusual point in this definition is that variables are arrays $x[\widetilde{i}]$. We extend this definition to terms and processes in which the replication indices $\widetilde{i}$ have been instantiated to bitstrings: for example, $\mathsf{fv}(x[\widetilde{a}]) = \{x[\widetilde{a}]\}$. We extend this definition to sets of processes by $\mathsf{fv}(\mathcal{Q}) = \bigcup_{Q \in \mathcal{Q}} \mathsf{fv}(Q)$ and to stacks by $\mathsf{fv}(\mathcal{R}) = \bigcup_{((x_1[\widetilde{a}],\ldots,x_k[\widetilde{a}]),P_1,P_2) \in \mathcal{R}} \mathsf{fv}(P_1) \setminus \{x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]\} \cup \mathsf{fv}(P_2)$.

Next, we define the OCaml value corresponding to a CryptoVerif table, and we use this definition to define the OCaml environment and global store corresponding to a CryptoVerif environment and to CryptoVerif tables.

Definition 8.11 (CryptoVerif table to OCaml list) *Let us consider a table Tbl of type $T_1 \times \cdots \times T_l$. The serialized OCaml value that corresponds to an element of this table is*

$$\mathbb{G}_{\mathsf{tblel}}(Tbl, (b_1, \ldots, b_l)) \stackrel{\text{def}}{=} (\mathsf{ser}(T_1, \mathbb{G}_{\mathsf{val}T_1}(b_1)), \ldots, \mathsf{ser}(T_l, \mathbb{G}_{\mathsf{val}T_l}(b_l))) \,.$$

Let $t = [a_1; \ldots; a_k]$ be the contents of the table Tbl: each a_i is an element of the table. Let us denote

$$\mathbb{G}_{\mathsf{tbl}}(Tbl, t) \stackrel{\text{def}}{=} [\mathbb{G}_{\mathsf{tblel}}(a_1); \ldots; \mathbb{G}_{\mathsf{tblel}}(a_k)]$$

the OCaml list corresponding to t.

Definition 8.12 (Minimal environment and global store)

$$env(E, P) \stackrel{\text{def}}{=} \{\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T_x}(b) \mid x[\widetilde{a}] \in \mathsf{fv}(P), E(x[\widetilde{a}]) = b\} \quad \text{(Environment)}$$

$$\begin{aligned} globalstore(E, \mathcal{T}) \stackrel{\text{def}}{=} & \{f \mapsto \mathbb{G}_{\mathsf{tbl}}(Tbl, \mathcal{T}(Tbl)) \mid (Tbl, f) \in \mathsf{tables}\} \\ & \cup \{f \mapsto \mathsf{ser}(T_x, a) \mid (x[\,], f) \in \mathsf{files}, E(x[\,]) = a\} \\ & \cup \{f \mapsto \texttt{""} \mid (x[\,], f) \in \mathsf{files}, x \textit{ not defined in } E\} \end{aligned} \quad \text{(Globalstore)}$$

We define $env(E, M)$ and $env(E, Q)$ in the same way.

The *globalstore* function defined above returns the global store in which the contents of the files and the tables is correct with respect to the CryptoVerif configuration elements E and $\mathcal{T}$. The *env* function returns the environment corresponding to E for the free variables in P (or M, or Q).

First, we show a correspondence between a CryptoVerif term and its OCaml translation.

Lemma 8.13 (Term reduction) *Let M be a CryptoVerif term of type T. If*

$$Th = \langle env, \mathbb{G}_{\mathsf{M}}(M), stack, store\rangle \textit{ with } env \supseteq env_{\mathsf{prim}} \cup env(E, M) \,,$$

and $E, M \Downarrow a$, then $Th \rightarrow^* Th'$ where

$$Th' \stackrel{\text{def}}{=} \langle env', \mathbb{G}_{\mathsf{val}T}(a), stack, store'\rangle$$

for some env' and $store'$ such that $store' \supseteq store$.

Proof We prove this result by induction on the syntax of terms.

- Case $M = x[\widetilde{a'}]$: Since $E, M \Downarrow a$ is derived by (Var), we have $E(x[\widetilde{a'}]) = a$. Since $env(E, M) \subseteq env$, we have $env(\mathbb{G}_{\mathsf{var}}(x)) = \mathbb{G}_{\mathsf{val}T}(a)$, so

$$Th = \langle env, \mathbb{G}_{\mathsf{var}}(x), stack, store\rangle \to Th' = \langle env, \mathbb{G}_{\mathsf{val}T}(a), stack, store\rangle$$

- Case $M = f(M_1, \ldots, M_k)$, where f is of type $T_1 \times \cdots \times T_k \to T$. Since $E, M \Downarrow a$ is derived by (Fun), we have $E, M_i \Downarrow a_i$ for all $i \leq k$, for some $a_1, \ldots, a_k$ such that $f(a_1, \ldots, a_k) = a$.

$$\begin{aligned}
Th &= \langle env, \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_k)), stack, store\rangle \\
&\to \langle env, (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_k)), stack', store\rangle \\
&\quad \text{where } stack' \stackrel{\text{def}}{=} (env, \mathbb{G}_{\mathsf{f}}(f)\ [\cdot]) :: stack \\
&\to Th_1 \stackrel{\text{def}}{=} \langle env, \mathbb{G}_{\mathsf{M}}(M_k), stack'', store\rangle \\
&\quad \text{where } stack'' \stackrel{\text{def}}{=} (env, (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{k-1}), [\cdot])) :: stack' \\
&\to^* Th_2 \stackrel{\text{def}}{=} \langle env', \mathbb{G}_{\mathsf{val}T_k}(a_k), stack'', store'\rangle \\
&\qquad\qquad \text{by induction hypothesis applied to } M_k \\
&\to \langle env, (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{k-1}), \mathbb{G}_{\mathsf{val}T_k}(a_k)), stack', store'\rangle \\
&\to^* Th_3 \stackrel{\text{def}}{=} \langle env, (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), stack', store''\rangle \\
&\qquad\qquad \text{by an easy induction} \\
&\to \langle env, \mathbb{G}_{\mathsf{f}}(f)\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), stack, store''\rangle \\
&\to \langle env, \mathbb{G}_{\mathsf{f}}(f), stack''', store''\rangle \\
&\quad \text{where } stack''' \stackrel{\text{def}}{=} (env, [\cdot]\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k))) :: stack \\
&\to \langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{f}}(f)), stack''', store''\rangle \qquad \text{since } env_{\mathsf{prim}} \subseteq env \\
&\to \langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{f}}(f))\ (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k)), stack, store''\rangle \\
&\to^* Th' \stackrel{\text{def}}{=} \langle env'', \mathbb{G}_{\mathsf{val}T}(a), stack, store'''\rangle \qquad \text{by Proposition 8.5}
\end{aligned}$$

By Proposition 8.5 and induction hypothesis, we have $store''' \supseteq store'' \supseteq store' \supseteq store$. □

Let us now introduce some notations.

Definition 8.14 (Helper functions) *For an OCaml configuration*

$$\mathcal{C} = [Th_1, \ldots, Th_n], globalstore, tj', \mathcal{MI}, events$$

with $Th_{tj} = \langle env_{tj}, pe_{tj}, stack_{tj}, store_{tj}\rangle$ for all $tj \leq n$, let us define the following functions:

- $\mathcal{C}_{pe}(\mathcal{C}, tj) \stackrel{\text{def}}{=} pe_{tj}$, $\quad \mathcal{C}_{pe}(\mathcal{C}) \stackrel{\text{def}}{=} \mathcal{C}_{pe}(\mathcal{C}, tj')$,
- $\mathcal{C}_{Th}(\mathcal{C}, tj) \stackrel{\text{def}}{=} Th_{tj}$, $\quad \mathcal{C}_{Th}(\mathcal{C}) \stackrel{\text{def}}{=} \mathcal{C}_{Th}(\mathcal{C}, tj')$,
- $\mathcal{C}_t(\mathcal{C}) \stackrel{\text{def}}{=} tj'$,
- $\mathcal{C}_{tmax}(\mathcal{C}) \stackrel{\text{def}}{=} n$,

- $\mathcal{C}_{globalstore}(\mathcal{C}) \stackrel{\text{def}}{=} globalstore$,
- $\mathcal{C}_{events}(\mathcal{C}) \stackrel{\text{def}}{=} events$,

We also define these functions on traces, where they return the elements in the last configuration of the trace.

We also define

$$\mathcal{C}[\mathsf{Th} \mapsto Th', \mathsf{globalstore} \mapsto globalstore', \mathsf{events} \mapsto events', \mathsf{MI} \mapsto \mathcal{MI}'] \stackrel{\text{def}}{=} [Th_1, \ldots, Th_{tj'-1}, Th', Th_{tj'+1}, \ldots, Th_n], globalstore', tj', \mathcal{MI}', events' .$$

In this notation, one can omit globalstore, events, *or* MI. *When omitted, we keep the corresponding element of the configuration* $\mathcal{C}$.

Next, we prove that the CryptoVerif oracle bodies P are correctly translated into OCaml as $\mathbb{G}(P)$. We extend the translation $\mathbb{G}(P)$ to processes in which some replication indices have been instantiated into their values, using the formulas of Chapters 3 and 7 where replication indices i may be replaced with their value a. It is easy to see that $\mathbb{G}(P\{a/i\}) = \mathbb{G}(P)$: we do not use the values of the indices in the translation.

Lemma 8.15 (Inner reduction) *Let* $\mathfrak{C}$ *be a CryptoVerif configuration. Suppose that the program part* P *of* $\mathfrak{C}$ *is not in a return, end, call, or loop form. Suppose that we have* n *possible reductions beginning at this configuration:*

$$\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_{p_i} \mathfrak{C}_i = E_i, P_i, \mathcal{Q}, \mathcal{T}_i, \mathcal{R}, \mathcal{E}_i$$

for $i \leq n$. *Let* $\mathcal{C}$ *be an OCaml configuration such that*

$$\begin{aligned} \mathcal{C}_{Th}(\mathcal{C}) &= \langle env, \mathbb{G}(P), stack, store\rangle \text{ with } env \supseteq env_{\mathsf{prim}} \cup env(E, P) , \\ \mathcal{C}_{globalstore}(\mathcal{C}) &\supseteq globalstore(E, \mathcal{T}) , \\ \mathcal{C}_{events}(\mathcal{C}) &= \mathbb{G}_{\mathsf{ev}}(\mathcal{E}) . \end{aligned}$$

Then there exist n *disjoint sets of OCaml traces* $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ *all starting at* $\mathcal{C}$ *such that none of these traces is a prefix of another of these traces,* $\Pr[\mathcal{CTS}_i] = p_i$ *for all* $i \leq n$, *and if* $\mathcal{C}'$ *is the last configuration of a trace in* $\mathcal{CTS}_i$, *then we have* $\mathcal{C}' = \mathcal{C}[\mathsf{Th} \mapsto Th', \mathsf{globalstore} \mapsto globalstore', \mathsf{events} \mapsto events']$ *where*

$$\begin{aligned} Th' &= \langle env', \mathbb{G}(P_i), stack, store'\rangle \\ &\quad \text{with } env' \supseteq env_{\mathsf{prim}} \cup env(E_i, P_i) \text{ and } store' \supseteq store , \\ globalstore' &\supseteq globalstore(E_i, \mathcal{T}_i) , \\ globalstore'(l) &= \mathcal{C}_{globalstore}(\mathcal{C})(l) \text{ for all } l \notin S_{priv} , \\ events' &= \mathbb{G}_{\mathsf{ev}}(\mathcal{E}_i) . \end{aligned}$$

The proof of this lemma can be found in Appendix A. This lemma is proved by cases on the process P. We use Lemma 8.13 when we need to evaluate a term. The cases end and return will be handled when we prove the invariant for the whole system. The oracle bodies that we translate into OCaml do not contain calls nor loops. This lemma shows that the following invariants are

preserved during the evaluation of oracle bodies: the OCaml environment and global store contain the minimal environment and global store corresponding to the CryptoVerif configuration; the public part of the global store does not change; the OCaml and CryptoVerif events match. Locations may be added in the store, but the contents of existing locations does not change.

8.3 Simulation of the OCaml Adversary

In this section, we show how to simulate in CryptoVerif any OCaml program $program_0$ that corresponds to an adversary interacting with the protocol implementation generated from the CryptoVerif process Q_0. Basically, we run the OCaml program $program_0$ inside the CryptoVerif primitive *simulate* (which is possible since these primitives can represent any deterministic Turing machine). When $program_0$ needs to call an oracle of Q_0, the primitive returns and the call is made by CryptoVerif. When $program_0$ needs to generate a random number, this generation is performed by CryptoVerif.

We assume that the OCaml program $program_0$ runs in bounded time, so makes a bounded number of oracle calls. By Assumption 5.5, when an oracle O (resp. role role) is under replication, this replication has bound N_O (resp. N_{role}). When oracle O is under replication, we let N_O be the maximum number of calls to the same closure $\mathsf{tagfunction}^{O,\tau}[env, pm]$ corresponding to oracle O. When a role role is under replication, we let N_{role} be the maximum number of executions of $\mathsf{addthread}(program)$ for some $program$ that contains μ_{role}. These replication bounds are chosen such that the OCaml program $program_0$ never exhausts the number of oracle calls allowed by the CryptoVerif process. We let $N_{\mathsf{rand+calls}}$ be the maximum number of oracle calls and random number generations that the OCaml program $program_0$ can make plus one. We let N_{steps} be the maximum number of reduction steps of the program $program_0$ in the semantics of OCaml. Formally, we use the following definition:

Definition 8.16 *The number of calls to the closure with tag O, τ in a trace $\mathcal{CT}$, denoted $N_{\mathsf{calls}}(O, \tau, \mathcal{CT})$, is the number of configurations $\mathcal{C}$ such that $\mathcal{C}_{pe}(\mathcal{C}) = \mathsf{tagfunction}^{O,\tau}[env, pm]$ v in $\mathcal{CT}$ excluding its last configuration.*

The number of executions of role role in a trace $\mathcal{CT}$, denoted $N_{\mathsf{exec}}(\mathsf{role}, \mathcal{CT})$, is the number of configurations $\mathcal{C}$ such that $\mathcal{C}_{pe}(\mathcal{C}) = \mathsf{addthread}(program)$ where $program$ contains $program(\mu_{\mathsf{role}})$ in $\mathcal{CT}$ excluding its last configuration.

The number of random number generations in a trace $\mathcal{CT}$, denoted $N_{\mathsf{rand}}(\mathcal{CT})$, is the number of transitions derived by rule (Random) *in $\mathcal{CT}$.*

We define

$$N_O \stackrel{\text{def}}{=} \max_{\mathcal{CT},\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT})$$

$$N_{\mathsf{role}} \stackrel{\text{def}}{=} \max_{\mathcal{CT}} N_{\mathsf{exec}}(\mathsf{role}, \mathcal{CT})$$

$$N_{\mathsf{rand+calls}} \stackrel{\text{def}}{=} \max_{\mathcal{CT}} \left(N_{\mathsf{rand}}(\mathcal{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT}) \right) + 1$$

$$N_{\mathsf{steps}} \stackrel{\text{def}}{=} \max_{\mathcal{CT}} |\mathcal{CT}|$$

Figure 8.2 The program $Q_{\text{adv}}(Q_0, \mathit{program}_0)$

$Q_{\text{adv}}(Q_0, \mathit{program}_0) = Q_{\text{start}}(Q_0, \mathit{program}_0) \mid Q_c(Q_0, \mathit{program}_0)$
$Q_{\text{start}}(Q_0, \mathit{program}_0) = O_{\text{start}}() :=$
  $s_0 : T_{\mathcal{CS}} \leftarrow s_0(Q_0, \mathit{program}_0);$
  let $r : T_{\mathcal{CS}} =$ loop $O_{\text{loop}}(s_0)$ in end else end
$Q_c(Q_0, \mathit{program}_0) =$ foreach $i' \leq N_{\mathsf{rand+calls}}$ do
  $O_{\text{loop}}[i'](s : T_{\mathcal{CS}}) :=$
    let $(s' : T_{\mathcal{CS}}, o : T_o, i : \mathit{bitstring}, \mathit{args} : \mathit{bitstring}) = \mathit{simulate}(s)$ in
    if $o = o_S$ then
      return(s', false)
    else if $o = o_1$ then
      let $(a_{1,1} : T_{1,1}, \ldots, a_{1,m_1} : T_{1,m_1}) = \mathit{args}$ in
      let $(i_{1,1} : [1, N_{1,1}], \ldots, i_{1,n_1} : [1, N_{1,n_1}]) = i$ in
      let $(r_{1,1} : T'_{1,1}, \ldots, r_{1,m'_1} : T'_{1,m'_1}) = O_1[i_{1,1}, \ldots, i_{1,n_1}](a_{1,1}, \ldots, a_{1,m_1})$ in
        return$(\mathit{simulate}'_{O_1}(s', (r_{1,1}, \ldots, r_{1,m'_1})), \text{true})$
      else return$(\mathit{simulate}''_{O_1}(s'), \text{true})$
    else if $o = o_2$ then
      $\vdots$
    else if $o = o_R$ then
      $b_R \xleftarrow{R} \mathit{bool};$
      return$(\mathit{simulate}_R(s', b_R), \text{true})$

where $\mathcal{CT}$ ranges over traces that begin with the configuration $\mathcal{C}_0(Q_0, \mathit{program}_0)$.

While N_O is an optimal bound, N_{role} is not optimal. Consider for instance a process of the form

$$\mathsf{foreach}\ i \leq N_O\ \mathsf{do}\ O() := \ldots\}\ \mathsf{foreach}\ j \leq N_{\mathsf{role}}\ \mathsf{do}\ \mathsf{role}\ \{\ldots$$

By distributing the instantiations of role on every available index i, the optimal bound of the replication j is the maximum during all executions of $\mathit{program}_0$ of the number of instantiations of role divided by the number of calls to O made before these instantiations of role. To get this optimal bound, we would need to associate each new instantiation of role to the index i with the least number of associated instantiations of role. Since a role is often under at most one replication, we decided not to complicate the proof with details needed to get the optimal bound.

From the OCaml program $\mathit{program}_0$, we define a CryptoVerif adversary $Q_{\text{adv}}(Q_0, \mathit{program}_0)$ given in Figure 8.2 and explained below. We will prove that this process, when executed in parallel with Q_0, has the same behavior as the OCaml program $\mathit{program}_0$. The initial CryptoVerif configuration is then $\mathfrak{C}_0(Q_0, \mathit{program}_0) = \mathfrak{C}_{\text{i}}(Q_0 \mid Q_{\text{adv}}(Q_0, \mathit{program}_0))$.

In Figure 8.2, we use a let construct with pattern matching, which can be defined as follows. We define the function $\mathit{tuple}_{T_1,\ldots,T_j} : T_1 \times \cdots \times T_j \to \mathit{bitstring}$

that creates a tuple with j elements (for instance by concatenating the j bitstrings with information on their length, so that they can be unambiguously recovered), and the associated projections $\pi_{k,T_1,\ldots,T_j} : \mathit{bitstring} \to T_k$ with $k \leq j$ (which may return any value when their argument is not a tuple with j elements). The construct $\mathsf{let}\ (x_1 : T_1, \ldots, x_j : T_j) = M\ \mathsf{in}\ P$ is an abbreviation for:

$$x \leftarrow M; x_1 \leftarrow \pi_{1,T_1,\ldots,T_j}(x); \ldots; x_j \leftarrow \pi_{j,T_1,\ldots,T_j}(x);\\ \mathsf{if}\ x = \mathit{tuple}_{T_1,\ldots,T_j}(x_1, \ldots, x_j)\ \mathsf{then}\ P$$

where x is a fresh variable. The CryptoVerif term $(M_1, \ldots, M_j)$ is an abbreviation for $\mathit{tuple}_{T_1,\ldots,T_j}(M_1, \ldots, M_j)$, where $T_1, \ldots, T_j$ are the types of $M_1, \ldots, M_j$, respectively.

Let $O_1, \ldots, O_n$ be the oracle names in Q_0. We define n constants $o_1, \ldots, o_n$ which are used to designate the oracles $O_1, \ldots, O_n$ respectively, o_R which corresponds to a random choice, and o_S which corresponds to the end of the OCaml program. We define the CryptoVerif type $T_o \stackrel{\text{def}}{=} \{o_R, o_S, o_1, \ldots, o_n\}$, which contains all these bitstring constants.

The adversary is mainly encoded by the function *simulate*. This function takes as argument the bitstring representation $s = \mathsf{repr}(\mathcal{CS})$ of a simulator configuration $\mathcal{CS}$. The configuration $\mathcal{CS}$ consists of a non-instrumented OCaml configuration $\mathcal{C}$ (with some extensions to the syntax described later) and sets $\mathcal{RI}$ and $\mathcal{I}$ that finitely represent the callable oracles $\mathcal{Q}$ of the CryptoVerif configuration:

$$\mathcal{CS} = \underbrace{([\mathit{Th}_1, \ldots, \mathit{Th}_n], \mathit{globalstore}, i)}_{\mathcal{C}}, \mathcal{RI}, \mathcal{I}\,.$$

The function repr is injective. We denote its inverse by repr^{-1}. We also define a CryptoVerif type $T_{\mathcal{CS}}$ that consists of all bitstrings in the image of repr, that is, all bitstrings that correspond to simulator configurations $\mathcal{CS}$. We also use the notations of Definition 8.14 for simulator configurations.

When we call an oracle or instantiate a role under replication, we must choose an unused replication index for this replication, and call the oracle or instantiate the role with that replication index. In this simulation, we will always choose the smallest replication index that has not been used yet, so that the used indices form an interval $[1, a-1]$ and the unused indices are in $[a, N]$ where N is the bound of the considered replication. The sets $\mathcal{RI}$ and $\mathcal{I}$ represent the sets of callable roles and oracles, by storing the smallest index a that is not used yet.

More precisely, the set $\mathcal{RI}$ represents the set of callable roles with their replication indices. Elements of $\mathcal{RI}$ are either:

- of the form $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$, which means the role role is under replication, the roles $\mathsf{role}[1, \widetilde{a'}]$ to $\mathsf{role}[a-1, \widetilde{a'}]$ have been used, and the roles $\mathsf{role}[a, \widetilde{a'}]$ to $\mathsf{role}[N_{\mathsf{role}}, \widetilde{a'}]$ are usable,

- or of the form $\mathsf{role}[\widetilde{a}]$, which means that role is not under replication and the role role is callable with the replication indices $\widetilde{a}$.

The set $\mathcal{RI}$ never contains simultaneously $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$ and $\mathsf{role}[a'', \widetilde{a'}]$ for the same role and $\widetilde{a'}$, and it never contains simultaneously $\mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big]$ and $\mathsf{role}\big[[a'', +\infty[, \widetilde{a'}\big]$ with $a \neq a''$ for the same role and $\widetilde{a'}$.

The set $\mathcal{I}$ represents the set of callable oracles with their replication indices. Elements of $\mathcal{I}$ are either:

- of the form $O\big[[a, +\infty[, \widetilde{a'}\big]$, which means that the oracle O is under replication and the oracles $O[1, \widetilde{a'}]$ to $O[a-1, \widetilde{a'}]$ have been used, and the oracles $O[a, \widetilde{a'}]$ to $O[N_O, \widetilde{a'}]$ are usable,
- or of the form $O[\widetilde{a}]$ which means that O is an oracle not under replication that can be called with the replication indices $\widetilde{a}$.

The set $\mathcal{I}$ never contains simultaneously $O\big[[a, +\infty[, \widetilde{a'}\big]$ and $O[a'', \widetilde{a'}]$ for the same O and $\widetilde{a'}$, and it never contains simultaneously $O\big[[a, +\infty[, \widetilde{a'}\big]$ and $O\big[[a'', +\infty[, \widetilde{a'}\big]$ with $a \neq a''$ for the same O and $\widetilde{a'}$.

Next, we define functions that manipulate these sets of oracles and roles. We define the subtraction operation $\mathcal{I} - O[\widetilde{a}]$ on sets of oracles.

- If $O\big[[a, +\infty[, \widetilde{a'}\big]$ is in $\mathcal{I}$, then

$$\mathcal{I} - (O[a, \widetilde{a'}]) \stackrel{\text{def}}{=} \mathcal{I} \setminus \{O\big[[a, +\infty[, \widetilde{a'}\big]\} \cup \{O\big[[a+1, +\infty[, \widetilde{a'}\big]\}\,.$$

- If $O[\widetilde{a}]$ is in $\mathcal{I}$, then

$$\mathcal{I} - (O[\widetilde{a}]) \stackrel{\text{def}}{=} \mathcal{I} \setminus \{O[\widetilde{a}]\}\,.$$

We define similarly the subtraction on sets of roles $\mathcal{RI} - \mathsf{role}[\widetilde{a}]$. We also generalize this operator to sets:

$$\mathcal{RI} - \{\mathsf{role}_1[\widetilde{a_1}], \ldots, \mathsf{role}_k[\widetilde{a_k}]\} \stackrel{\text{def}}{=} (\ldots(\mathcal{RI} - \mathsf{role}_1[\widetilde{a_1}]) - \ldots) - \mathsf{role}_k[\widetilde{a_k}]\,.$$

We let $\mathsf{smallest}(\mathcal{RI}, \mathsf{role})$ be the smallest indices present for the role role in $\mathcal{RI}$: when $\widetilde{a} = \mathsf{smallest}(\mathcal{RI}, \mathsf{role})$, we have $\mathsf{role}[\widetilde{a}] \in \mathcal{RI}$ or there exist a' and $\widetilde{a'}$ such that $\widetilde{a} = a', \widetilde{a'}$ and $\mathsf{role}\big[[a', +\infty[, \widetilde{a'}\big] \in \mathcal{RI}$.

Let us define the function $\mathsf{oracles'}$, which is similar to the function reduce′, but just returns the oracle name and its replication indices $\widetilde{i}$ (which can be partly instantiated to values), instead of returning the entire oracle definition:

$$\mathsf{oracles'}(0) \stackrel{\text{def}}{=} [\,] \qquad \text{(Nil)}$$

$$\mathsf{oracles'}(Q_1 \mid Q_2) \stackrel{\text{def}}{=} \mathsf{oracles'}(Q_1) \mathbin{@} \mathsf{oracles'}(Q_2) \qquad \text{(Par)}$$

$$\mathsf{oracles'}(\mathsf{foreach}\ i' \leq n\ \mathsf{do}\ Q) \stackrel{\text{def}}{=} \big[O_1[\_, \widetilde{i}], \ldots, O_l[\_, \widetilde{i}]\big] \text{ when } \mathsf{oracles'}(Q) = \big[O_1[i', \widetilde{i}], \ldots, O_l[i', \widetilde{i}]\big] \qquad \text{(Repl)}$$

$$\mathsf{oracles'}(\mathsf{role}\,\{Q) \stackrel{\text{def}}{=} [\,] \qquad \text{(Role)}$$

$$\mathsf{oracles'}(O[\widetilde{i}](x_1[\widetilde{i}], \ldots, x_k[\widetilde{i}]) := P) \stackrel{\text{def}}{=} \big[O[\widetilde{i}]\big] \qquad \text{(Oracle)}$$

This function returns elements of the form $O[\widetilde{i}]$ for oracles that are not directly under replication and $O[\_, \widetilde{i}]$ for oracles directly under replication. Similarly to reduce′, this function returns an empty list when encountering a role definition.

Let us consider a process $Q' = \mathsf{foreach}\ i' \leq n\ \mathsf{do}\ Q$. By Assumption 5.4, there is no replication in Q, and so all oracles in Q are under the same replications and have exactly the same replication indices $i', \widetilde{i}$, where the indices $\widetilde{i}$ are the replication indices of replications above Q'. So, by rule (Repl), $\mathsf{oracles}'(Q')$ produces the list of callable oracles in Q where we replace the replication index i' with _.

By Property 1.4, an oracle with a certain name O always takes arguments of the same types and always returns values of the same types. So we can say that the oracle O_i takes m_i arguments of types $T_{i,1}, \ldots, T_{i,m_i}$, and returns m'_i bitstrings of types $T'_{i,1}, \ldots, T'_{i,m'_i}$. We can also define $\mathsf{returnoracles}(O[\widetilde{i}]) \stackrel{\text{def}}{=} \mathsf{oracles}'(Q)$ where Q is an oracle definition located after a return statement in a body of the oracle $O[\widetilde{i}]$ in Q_0. This definition is correct because, by Property 1.4, the structure of the processes Q after any return statement of a given oracle O is always the same, so the list $\mathsf{oracles}'(Q)$ will be the same for each of these Q. The function returnoracles can take an oracle with its replication indices partly instantiated to values: $\mathsf{returnoracles}(O[\widetilde{a}]) \stackrel{\text{def}}{=} \mathsf{returnoracles}(O[\widetilde{i}])\{\widetilde{a}/\widetilde{i}\}$.

Let us denote by $Q(\mathsf{role})$ the subprocess of Q_0 that corresponds to the role role. For a subprocess Q of Q_0 that is under replication indices $\widetilde{i}$ in Q_0, we denote $Q[\widetilde{a}]$ the process Q where we substituted elements of $\widetilde{i}$ by the respective elements of $\widetilde{a}$.

Definition 8.17 (First oracle) *The* first oracles *of a role* role *are the oracles that can be called when we are at the beginning of the subprocess corresponding to the role, that is,* $\mathsf{oracles}'(Q(\mathsf{role}))$.

We define $\mathsf{add}(\mathcal{I}, \mathcal{RI})$ as the addition of the first oracles present in $\mathcal{RI}$ to $\mathcal{I}$:

$$\mathsf{add}(\mathcal{I}, \mathcal{RI}) \stackrel{\text{def}}{=} \mathcal{I} \cup \{O[\widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[\widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])\} \cup$$
$$\{O\big[[1, +\infty[, \widetilde{a}\big] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[\_, \widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])\}$$

The syntax of the language of the simulator is almost the same as the language we described in Chapters 2 and 6, with the addition of tagged functions introduced in Section 6.2. We add the functional values $\mathsf{call}(O[\widetilde{a}])$ and $\mathsf{call}(O[\_, \widetilde{a}])$ that replace our generated closures for the oracle O. The value $\mathsf{call}(O[\widetilde{a}])$ is used when O is not directly under replication; $\mathsf{call}(O[\_, \widetilde{a}])$ is used when O is directly under replication.

We present the semantics followed by our simulator in Figure 8.3. When we encounter a configuration containing a successful call to an oracle (by call) or a random operation, we cannot reduce. These operations are executed, but not inside the simulator: we stop the simulator in its current state, and in CryptoVerif, we call the requested oracle with the requested arguments, or generate a random bit. Otherwise, when the simulator configuration reduces into another configuration in the OCaml semantics, by rule (Simulator toplevel), we also reduce in the same way. By rules (FailedCall1) and (FailedCall2), we raise the exception Bad_Call when the call to the oracle is invalid, as our

Figure 8.3 Semantics followed by the simulator

$$\frac{\begin{array}{l}O[\widetilde{a}] \notin \mathcal{I} \\ \text{or } O \text{ does not have } k \text{ arguments} \\ \text{or } O \text{ has } k \text{ arguments of type } T_1, \ldots, T_k \text{ and } \exists i,\ \nexists a_i, v_i = \mathbb{G}_{\mathsf{val}T_i}(a_i)\end{array}}{env, \mathsf{call}(O[\widetilde{a}])\ (v_1, \ldots, v_k), stack \longrightarrow env, \mathsf{raise\ Bad\_Call}, stack} \quad \text{(FailedCall1)}$$

$$\frac{\begin{array}{l}\forall a', O\big[[a', +\infty[, \widetilde{a}\big] \notin \mathcal{I} \\ \text{or } O \text{ does not have } k \text{ arguments} \\ \text{or } O \text{ has } k \text{ arguments of type } T_1, \ldots, T_k \text{ and } \exists i,\ \nexists a_i, v_i = \mathbb{G}_{\mathsf{val}T_i}(a_i)\end{array}}{env, \mathsf{call}(O[\_, \widetilde{a}])\ (v_1, \ldots, v_k), stack \longrightarrow env, \mathsf{raise\ Bad\_Call}, stack} \quad \text{(FailedCall2)}$$

$$\frac{\begin{array}{l}\mathcal{C} \to \mathcal{C}' \text{ using the rules of Figures 6.1–6.8, (FailedCall1), and (FailedCall2)} \\ \qquad \text{but not (Random) and (Toplevel add thread)}\end{array}}{\mathcal{C}, \mathcal{RI}, \mathcal{I} \to \mathcal{C}', \mathcal{RI}, \mathcal{I}} \quad \text{(Simulator toplevel)}$$

$$\frac{\begin{array}{l}program^a = program_{\mathsf{prim}};; program(\mu_{\mathsf{role}_1});; \ldots;; program(\mu_{\mathsf{role}_l});; program' \\ program' \text{ does not contain } program(\mu_{\mathsf{prim}}) \text{ nor any } program(\mu) \text{ for } \mu \in \mathcal{M}_g \\ \{\mu_{\mathsf{role}_1}, \ldots, \mu_{\mathsf{role}_l}\} \subseteq \mathcal{M}_g \\ \widetilde{a_1} = \mathsf{smallest}(\mathcal{RI}, \mathsf{role}_1), \ldots, \widetilde{a_l} = \mathsf{smallest}(\mathcal{RI}, \mathsf{role}_l) \\ \mathcal{RI}'' = \{\mathsf{role}_1[\widetilde{a_1}], \ldots, \mathsf{role}_l[\widetilde{a_l}]\} \qquad \mathcal{RI}' = \mathcal{RI} - \mathcal{RI}'' \qquad \mathcal{I}' = \mathsf{add}(\mathcal{I}, \mathcal{RI}'') \\ program^b = program_{\mathsf{prim}};; program'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots;; program'(\mathsf{role}_l[\widetilde{a_l}]);; \\ \qquad program' \\ \text{or} \\ program^a \text{ does not contain } program(\mu_{\mathsf{prim}}) \text{ nor any } program(\mu) \text{ for } \mu \in \mathcal{M}_g \\ \mathcal{RI}'' = \emptyset \qquad \mathcal{RI}' = \mathcal{RI} \qquad \mathcal{I}' = \mathcal{I} \qquad program^b = program^a\end{array}}{\begin{array}{l}[Th_1, \ldots, Th_{tj-1}, \langle env, \mathsf{addthread}(program^a), stack, store\rangle, Th_{tj+1}, \ldots, Th_n], \\ \quad globalstore, tj, \mathcal{RI}, \mathcal{I} \longrightarrow \\ [Th_1, \ldots, Th_{tj-1}, \langle env, (), stack, store\rangle, Th_{tj+1}, \ldots, Th_n, \langle \emptyset, program^b, [\,], \emptyset\rangle], \\ \quad globalstore, tj, \mathcal{RI}', \mathcal{I}'\end{array}} \quad \text{(Simulator add thread)}$$

generated code does in this case. Notice that, in the OCaml implementation, the adversary can test whether an oracle call succeeds or not, by catching the exception Bad_Call. In CryptoVerif, failed calls can happen only when the called oracle is not available, and in this case, the reduction blocks. This different behavior does not give additional power to the OCaml adversary, because the adversary can test before performing the call whether it will succeed or not. The rules (FailedCall1) and (FailedCall2) implement this test. By rule (Simulator add thread), we modify the behavior of the addthread construct to transform references to our generated modules $program(\mu_{\mathsf{role}})$ into references to the corresponding role $program'(\mathsf{role}[\widetilde{a}])$ where $\widetilde{a}$ are the replication indices we chose for this particular reference and

$$program'(\mathsf{role}[\widetilde{a}]) \stackrel{\text{def}}{=} \mathsf{let}\ \mu_{\mathsf{role}}.\mathsf{init} = \mathsf{let}\ token = \mathsf{ref}\ \mathrm{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}}\ pm'_{\mathsf{role}[\widetilde{a}]}$$
$$\text{where } pm'_{\mathsf{role}[\widetilde{a}]} \stackrel{\text{def}}{=} () \rightarrow$$
$$\mathsf{if}\ (!token)\ \mathsf{then}\ (token := \mathrm{false}; (\mathsf{call}(O_1[\widetilde{a_1}]), \ldots, \mathsf{call}(O_k[\widetilde{a_k}])))$$
$$\mathsf{else}\ \mathsf{raise}\ \mathsf{Bad\_Call}$$

where $\mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}]) = [O_1[\widetilde{a_1}], \ldots, O_k[\widetilde{a_k}]]$, and the $\widetilde{a_j}$ are either $\widetilde{a}$ or $\_, \widetilde{a}$. In particular, the initialization function defined in $program'(\mathsf{role}[\widetilde{a}])$ returns oracles represented by call values instead of closures.

The CryptoVerif function $simulate : T_{\mathcal{CS}} \rightarrow bitstring$ follows the simulator semantics defined in Figure 8.3: we define $simulate(\mathsf{repr}(\mathcal{CS})) \stackrel{\text{def}}{=} simreturn(\mathcal{CS}')$ where $\mathcal{CS}'$ is the configuration such that either $\mathcal{CS}$ reduces into $\mathcal{CS}'$ in at most N_{steps} reductions and $\mathcal{CS}'$ does not reduce, or $\mathcal{CS}$ reduces into $\mathcal{CS}'$ in exactly N_{steps} reductions, by the semantics of Figure 8.3, and $simreturn(\mathcal{CS}')$ is defined below. (We need to bound the number of reductions to make sure that *simulate* is always defined. The proof of the simulation between OCaml and CryptoVerif, presented in the next section, shows that the simulator configuration always blocks after at most N_{steps} reductions, so that we are always in the first case.)

- If $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{call}(O[\widetilde{a}])\ (v_1, \ldots, v_l)$, let $T_1, \ldots, T_l$ be the type of the arguments of the oracle O and let o be the constant associated to O. We define
$$simreturn(\mathcal{CS}') \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}'), o, \widetilde{a}, (\mathbb{G}^{-1}_{\mathsf{val}T_1}(v_1), \ldots, \mathbb{G}^{-1}_{\mathsf{val}T_l}(v_l)))\,.$$
- If $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{call}(O[\_, \widetilde{a}])\ (v_1, \ldots, v_l)$, let $T_1, \ldots, T_l$ be the type of the arguments of the oracle O, let o be the constant associated to O, and let a' be the value such that $O\big[[a', +\infty[, \widetilde{a}\big]$ is in the set $\mathcal{I}$ where $\mathcal{CS}' = \mathcal{C}, \mathcal{RI}, \mathcal{I}$. We define
$$simreturn(\mathcal{CS}') \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}'), o, (a', \widetilde{a}), (\mathbb{G}^{-1}_{\mathsf{val}T_1}(v_1), \ldots, \mathbb{G}^{-1}_{\mathsf{val}T_l}(v_l)))\,.$$
- If $\mathcal{C}_{pe}(\mathcal{CS}') = \mathsf{random}\ ()$, we define
$$simreturn(\mathcal{CS}') \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}'), o_R, (), ())\,.$$
- Otherwise, we define
$$simreturn(\mathcal{CS}') \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}'), o_S, (), ())\,.$$

The function *simulate* can be implemented by a deterministic Turing machine (since the random choices are handled outside *simulate*), so it can be used as a CryptoVerif primitive.

When *simulate* returns $(\mathsf{repr}(\mathcal{CS}'), o, \widetilde{a}, (a_1, \ldots, a_l))$, the CryptoVerif process $Q_c(Q_0, \mathit{program}_0)$ performs the corresponding oracle call $O[\widetilde{a}](a_1, \ldots, a_l)$ (lines 10–17 of Figure 8.2). Similarly, when *simulate* returns $(\mathsf{repr}(\mathcal{CS}'), o_R, (), ())$, the process $Q_c(Q_0, \mathit{program}_0)$ performs a random choice (lines 18–20), and when *simulate* returns $(\mathsf{repr}(\mathcal{CS}'), o_S, (), ())$, the process $Q_c(Q_0, \mathit{program}_0)$ terminates (lines 8–9; the corresponding OCaml program also terminates).

The functions $\mathit{simulate}'_O$ and $\mathit{simulate}''_O$ replace, in the simulator configuration, the call expression with the result returned by the oracle, and raise the Match_failure exception, respectively. The function $\mathit{simulate}'_O$ handles the situation in which an oracle returns a result by return; the function $\mathit{simulate}''_O$ handles the situation in which the oracle terminates with end. Formally, these functions are defined as follows.

Definition 8.18 (Simulation of oracle return) *Let us consider a simulator configuration $\mathcal{CS} = \mathcal{C}, \mathcal{RI}, \mathcal{I}$, with*

$$\mathcal{C}_{pe}(\mathcal{CS}) = \mathsf{call}(O[\widetilde{a}])\ (v_1, \ldots, v_l) \text{ or } \mathsf{call}(O[\_, \widetilde{a'}])\ (v_1, \ldots, v_l)\,.$$

When $\mathcal{C}_{pe}(\mathcal{CS})$ is of the second form, we denote by $\widetilde{a}$ the indices $a'', \widetilde{a'}$ where a'' is such that $O\big[[a'', +\infty[, \widetilde{a'}\big] \in \mathcal{I}$. Let $\mathcal{I}' \stackrel{\text{def}}{=} \mathcal{I} - (O[\widetilde{a}])$.

We define the CryptoVerif function $\mathit{simulate}'_O : T_{\mathcal{CS}} \times \mathit{bitstring} \to T_{\mathcal{CS}}$ as follows.

If the returns in oracle O end the current role, then by Property 1.5, there is only one return *statement in O; let Q be the oracle definition following this statement, and let*

$$\begin{aligned}\mathcal{RI}' \stackrel{\text{def}}{=} &\{\mathsf{role}[\widetilde{a}] \mid (\mu_{\mathsf{role}}, \text{false}) \in \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q)\}\\ &\cup \{\mathsf{role}\big[[1, +\infty[, \widetilde{a}\big] \mid (\mu_{\mathsf{role}}, \text{true}) \in \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q)\}\,.\end{aligned}$$

Let $T_1, \ldots, T_n$ be the types of the return value of O. We define:

$$\mathit{simulate}'_O(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I}), (r_1, \ldots, r_n)) \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{C}', \mathcal{RI} \cup \mathcal{RI}', \mathcal{I}')$$

where $\mathcal{C}'$ is the configuration $\mathcal{C}$ in which the current expression is replaced with the translated result: $(\mathbb{G}_{\mathsf{val}T_1}(r_1), \ldots, \mathbb{G}_{\mathsf{val}T_n}(r_n))$.

If the returns in oracle O do not end the current role, then let us define $\mathcal{O} \stackrel{\text{def}}{=} \mathsf{returnoracles}(O[\widetilde{a}])$. Let $\mathcal{I}''$ be the set $\mathcal{I}'$ to which we added the oracles present in $\mathcal{O}$:

$$\mathcal{I}'' \stackrel{\text{def}}{=} \mathcal{I}' \cup \{O'\big[[1, +\infty[, \widetilde{a}\big] \mid O'[\_, \widetilde{a}] \in \mathcal{O}\} \cup \{O'[\widetilde{a}] \mid O'[\widetilde{a}] \in \mathcal{O}\}\,.$$

We define:

$$\mathit{simulate}'_O(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I}), (r_1, \ldots, r_n)) \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{C}', \mathcal{RI}, \mathcal{I}'')$$

where $\mathcal{C}'$ is the configuration $\mathcal{C}$ in which the current expression is replaced with the translated result: $(\mathsf{call}(O_1[\widetilde{a_1}]), \ldots, \mathsf{call}(O_l[\widetilde{a_l}]), \mathbb{G}_{\mathsf{val}T_1}(r_1), \ldots, \mathbb{G}_{\mathsf{val}T_n}(r_n))$, with $\mathcal{O} = \{O_1[\widetilde{a_1}], \ldots, O_l[\widetilde{a_l}]\}$ and the $\widetilde{a_j}$ are either $\widetilde{a}$ or $\_, \widetilde{a}$.

In all other cases (that is, $\mathcal{CS}$ is not of the form mentioned above or the bitstring a is not a tuple of n bitstrings of types $T_1, \ldots, T_n$), $simulate'_O(\mathsf{repr}(\mathcal{CS}), a)$ can take any value, since these cases are in fact not used.

Finally, we define the CryptoVerif function $simulate''_O : T_{\mathcal{CS}} \to T_{\mathcal{CS}}$ by:

$$simulate''_O(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I})) \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{C}'', \mathcal{RI}, \mathcal{I}')$$

where $\mathcal{C}''$ is the configuration $\mathcal{C}$ in which the current expression is replaced with raise Match_failure. *In all other cases (that is, $\mathcal{CS}$ is not of the form mentioned above), $simulate''_O(\mathsf{repr}(\mathcal{CS}))$ can take any value, since these cases are in fact not used.*

When the returns in oracle O end the current role, the function $simulate'_O$ does not return the oracles following the current oracle, but adds the corresponding roles to the role set $\mathcal{RI}$. The programs that contain these roles can then be launched by addthread.

Definition 8.19 (Random simulation) *We define the CryptoVerif function $simulate_R : T_{\mathcal{CS}} \times bool \to T_{\mathcal{CS}}$ by*

$$simulate_R(\mathsf{repr}(\mathcal{C}, \mathcal{RI}, \mathcal{I}), b) \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{C}'(b), \mathcal{RI}, \mathcal{I})$$

where $\mathcal{C}'(b)$ is the configuration $\mathcal{C}$ in which the current expression is replaced with the OCaml boolean value $\mathbb{G}_{\mathsf{val}\,bool}(b)$.

Let us finally define the initial state of the simulator. Let $\mathcal{RI}_0$ be the set of initially callable roles of Q_0 with their replication indices: $\mathcal{RI}_0 \stackrel{\text{def}}{=} \{\mathsf{role}[\,] \mid (\mu_{\mathsf{role}}, \text{false}) \in \mathbb{G}_{\mathsf{get},\mathcal{MI}}(Q_0)\} \cup \{\mathsf{role}[[1, +\infty[] \mid (\mu_{\mathsf{role}}, \text{true}) \in \mathbb{G}_{\mathsf{get},\mathcal{MI}}(Q_0)\}$. We define:

$$s_0(Q_0, program_0) \stackrel{\text{def}}{=} \mathsf{repr}(([\langle \emptyset, program_0, [\,], \emptyset \rangle], globalstore_0, 1), \mathcal{RI}_0, \emptyset)$$

8.4 Correspondence

In this section, we prove our main security theorem by relating the CryptoVerif and OCaml systems.

In CryptoVerif, the security properties are defined using distinguishers D which are functions that take a list of events $\mathcal{E}$ and return true or false. We denote by $\Pr[\mathfrak{C} : D]$ the probability of the set of complete CryptoVerif traces starting at $\mathfrak{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = \text{true}$.

For instance, to show that a protocol Q_0 satisfies a correspondence c of the form "for all a, if $e_1(a)$ has been executed, then $e_2(a)$ has also been executed", we define D_c by $D_c(\mathcal{E}) = \text{true}$ if and only if the correspondence does not hold, that is, $\mathcal{E}$ contains $e_1(a)$ but not $e_2(a)$ for some a. We can represent the adversary for Q_0 by any CryptoVerif process Q_{adv} that does not contain events nor variables that occur in Q_0. Then we bound the probability $\Pr[\mathfrak{C}_i(Q_0 \mid Q_{\text{adv}}) : D_c]$, that is, the probability that the adversary Q_{adv} breaks the correspondence in Q_0,

for any adversary Q_{adv} for Q_0. We can also define secrecy using events and distinguishers [18].

We use a similar definition in OCaml: $\Pr[\mathcal{C} : D]$ is the probability of the set of complete OCaml traces starting at $\mathcal{C}$ and such that the list of events *events* in their last configuration satisfies $D(\mathbb{G}_{\mathsf{ev}}^{-1}(\mathit{events})) = \text{true}$.

Our goal is to prove that, for all protocols Q_0, OCaml adversaries $\mathit{program}_0$, and distinguishers D, we have $\Pr[\mathfrak{C}_0(Q_0, \mathit{program}_0) : D] = \Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D]$. From this property, it is easy to see that, if CryptoVerif bounds the probability $\Pr[\mathfrak{C}_{\text{i}}(Q_0 \mid Q_{\text{adv}}) : D]$ for any adversary Q_{adv} for Q_0, then the same bound also holds for the probability $\Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D]$ corresponding to the generated implementation. Indeed, $\Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D] = \Pr[\mathfrak{C}_0(Q_0, \mathit{program}_0) : D] = \Pr[\mathfrak{C}_{\text{i}}(Q_0 \mid Q_{\text{adv}}(Q_0, \mathit{program}_0)) : D]$ and $Q_{\text{adv}}(Q_0, \mathit{program}_0)$ is an adversary for Q_0.

To that order, we first introduce an intermediate semantics for CryptoVerif that decomposes the evaluation of the function *simulate* into several small steps. We easily relate this semantics to the semantics of CryptoVerif. Next, in Section 8.4.2, we relate the intermediate semantics to the OCaml semantics. For this purpose, we introduce a relation between intermediate semantic configurations and OCaml traces, that, in particular, ensures that the events are the same on both sides and we prove that this relation is preserved by reduction. Finally, in Section 8.4.3, we use these results to prove our main theorem.

8.4.1 Intermediate Semantics

We introduce extended CryptoVerif configurations $\mathfrak{C}^{\mathsf{cs}}$, which are configurations of the form $\mathfrak{C}$ or $\mathfrak{C}, \mathit{steps}, \mathcal{CS}$, where $\mathcal{CS}$ is a simulator configuration and *steps* is the maximum number of reductions of $\mathcal{CS}$ that can still be performed. (We use the field *steps* to guarantee termination.) The configurations $\mathfrak{C}, \mathit{steps}, \mathcal{CS}$ serve to represent the state of the system during the evaluation of the function *simulate*. We define a reduction relation $\rightsquigarrow$ on the extended configurations $\mathfrak{C}^{\mathsf{cs}}$.

Definition 8.20 *Let us define the reduction relation $\rightsquigarrow$ such that:*

$$\frac{E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_p \mathfrak{C}' \quad P \text{ is not of the form } x[a] \leftarrow \mathit{simulate}(s[a]); P' \text{ for any } x, a, P'}{E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightsquigarrow_p \mathfrak{C}'} \quad \text{(CryptoVerif)}$$

$$\frac{E(s[a]) = \mathsf{repr}(\mathcal{CS})}{\begin{array}{l}E, x[a] \leftarrow \mathit{simulate}(s[a]); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightsquigarrow \\ E, x[a] \leftarrow \mathit{simulate}(s[a]); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, N_{\mathsf{steps}}, \mathcal{CS}\end{array}} \quad \text{(Enter Simulator)}$$

$$\frac{\mathcal{CS} \rightarrow \mathcal{CS}' \qquad \mathit{steps} > 0}{E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{steps}, \mathcal{CS} \rightsquigarrow E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{steps} - 1, \mathcal{CS}'} \quad \text{(Simulator)}$$

$$\frac{\mathcal{CS} \text{ does not reduce or } \mathit{steps} = 0}{\begin{array}{l}E, x[a] \leftarrow \mathit{simulate}(s[a]); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{steps}, \mathcal{CS} \rightsquigarrow \\ E[x[a] \mapsto \mathit{simreturn}(\mathcal{CS})], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\end{array}} \quad \text{(Leave Simulator)}$$

When encountering a configuration $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ such that P is of the form $x[a] \leftarrow \mathit{simulate}(s[a]); P'$ and $E(s[a]) = \mathsf{repr}(\mathcal{CS})$, we reduce $\mathfrak{C}$ into an extended configuration $\mathfrak{C}, N_{\mathsf{steps}}, \mathcal{CS}$ by (Enter Simulator). We reduce $\mathcal{CS}$ by (Simulator) until it blocks or the number of allowed reductions N_{steps} is exhausted, and then we resume the CryptoVerif reductions by (Leave Simulator).

In the next lemma and proposition, we relate traces using $\rightsquigarrow$ to traces using $\rightarrow$, to prove that all events have the same probability in these two semantics.

Lemma 8.21 *Let $\mathfrak{C}$ be a CryptoVerif configuration.*

- *If $\mathfrak{C} \rightarrow_p \mathfrak{C}'$, then there is a trace $\mathfrak{C} \rightsquigarrow_p^* \mathfrak{C}'$ and all intermediate configurations in this trace (if any) are of the form $\mathfrak{C}, \mathit{steps}, \mathcal{CS}$.*
- *If $\mathfrak{C}$ does not reduce by $\rightarrow$, then it does not reduce by $\rightsquigarrow$ either.*

Proof Let us first show by induction on *steps* that, if $\mathcal{CS} \rightarrow^* \mathcal{CS}'$ in at most *steps* steps and $\mathcal{CS}'$ does not reduce, or $\mathcal{CS} \rightarrow^* \mathcal{CS}'$ in exactly *steps* steps, $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$, and $P = x[a] \leftarrow \mathit{simulate}(s[a]); P'$, then $\mathfrak{C}, \mathit{steps}, \mathcal{CS} \rightsquigarrow^* E[x[a] \mapsto \mathit{simreturn}(\mathcal{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$.

- If $\mathit{steps} = 0$ or $\mathcal{CS}$ does not reduce, then $\mathcal{CS}' = \mathcal{CS}$ and $\mathfrak{C}, \mathit{steps}, \mathcal{CS} \rightsquigarrow E[x[a] \mapsto \mathit{simreturn}(\mathcal{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$ by (Leave Simulator).
- If $\mathit{steps} > 0$ and $\mathcal{CS} \rightarrow \mathcal{CS}_1$, then $\mathcal{CS}_1 \rightarrow^* \mathcal{CS}'$ in at most $\mathit{steps} - 1$ steps and $\mathcal{CS}'$ does not reduce, or $\mathcal{CS}_1 \rightarrow^* \mathcal{CS}'$ in exactly $\mathit{steps} - 1$ steps, so by induction hypothesis,

  $$\mathfrak{C}, \mathit{steps} - 1, \mathcal{CS}_1 \rightsquigarrow^* E[x[a] \mapsto \mathit{simreturn}(\mathcal{CS})], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\,.$$

  By (Simulator), $\mathfrak{C}, \mathit{steps}, \mathcal{CS} \rightsquigarrow \mathfrak{C}, \mathit{steps} - 1, \mathcal{CS}_1$, so

  $$\mathfrak{C}, \mathit{steps}, \mathcal{CS} \rightsquigarrow^* E[x[a] \mapsto \mathit{simreturn}(\mathcal{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\,.$$

Let us now prove that, if $\mathfrak{C} \rightarrow_p \mathfrak{C}'$, then there is a trace $\mathfrak{C} \rightsquigarrow_p^* \mathfrak{C}'$ and all intermediate configurations in this trace (if any) are of the form $\mathfrak{C}, \mathit{steps}, \mathcal{CS}$. Let $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$.

- If P is not of the form $x[a] \leftarrow \mathit{simulate}(s[a]); P'$ for any x, a, P', then $\mathfrak{C} \rightsquigarrow_p \mathfrak{C}'$ by (CryptoVerif).
- If $P = x[a] \leftarrow \mathit{simulate}(s[a]); P'$ for some x, a, P', then by the semantics of CryptoVerif, $s[a] \in \mathrm{Dom}(E)$, $E(s[a])$ is of type $T_{\mathcal{CS}}$, and $\mathfrak{C}' = E[x[a] \mapsto \mathit{simulate}(E(s[a]))], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$. Since $E(s[a])$ is of type $T_{\mathcal{CS}}$, there exists a configuration $\mathcal{CS}$ such that $E(s[a]) = \mathsf{repr}(\mathcal{CS})$. By reduction rule (Enter Simulator), $\mathfrak{C} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} = \mathfrak{C}, N_{\mathsf{steps}}, \mathcal{CS}$. Moreover, by definition of *simulate*, $\mathcal{CS} \rightarrow^* \mathcal{CS}'$ in at most N_{steps} steps and $\mathcal{CS}'$ does not reduce, or $\mathcal{CS} \rightarrow^* \mathcal{CS}'$ in exactly N_{steps} steps, and $\mathit{simulate}(\mathsf{repr}(\mathcal{CS})) = \mathit{simreturn}(\mathcal{CS}')$. By the result shown above, $\mathfrak{C} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} \rightsquigarrow^* E[x[a] \mapsto \mathit{simreturn}(\mathcal{CS}')], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} = \mathfrak{C}'$.

Finally, let us show that, if $\mathfrak{C}$ does not reduce by $\rightarrow$, then it does not reduce by $\rightsquigarrow$ either. Let $\mathfrak{C} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$.

- If P is not of the form $x[a] \leftarrow \mathit{simulate}(s[a]); P'$ for any x, a, P', then the only rule applicable to reduce $\mathfrak{C}$ by $\rightsquigarrow$ is (CryptoVerif), and it cannot be applied because $\mathfrak{C}$ does not reduce by $\rightarrow$. Hence $\mathfrak{C}$ does not reduce by $\rightsquigarrow$.
- If $P = x[a] \leftarrow \mathit{simulate}(s[a]); P'$ for some x, a, P', then either $s[a] \notin \mathrm{Dom}(E)$ or $E(s[a]) \notin T_{\mathcal{CS}}$. The only rule applicable to reduce $\mathfrak{C}$ by $\rightsquigarrow$ is (Enter Simulator), and it does not apply when $s[a] \notin \mathrm{Dom}(E)$ or $E(s[a]) \notin T_{\mathcal{CS}}$. Hence $\mathfrak{C}$ does not reduce by $\rightsquigarrow$. (We could also show that, because the CryptoVerif configurations are well-typed, $\mathfrak{C}$ always reduces when $P = x[a] \leftarrow \mathit{simulate}(s[a]); P'$.) □

We denote by $\Pr[\mathfrak{C}^{\mathsf{cs}}(\rightsquigarrow) : D]$ the probability of the set of complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathfrak{C}^{\mathsf{cs}}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = \text{true}$. The next proposition shows that all events have the same probability in the intermediate semantics as in the CryptoVerif semantics.

Proposition 8.22 $\Pr[\mathfrak{C}(\rightsquigarrow) : D] = \Pr[\mathfrak{C} : D]$.

Proof (of Proposition 8.22) For $b \in \{\text{true}, \text{false}\}$, let $\mathfrak{CTS}_b$ be the set of complete CryptoVerif traces using $\rightarrow$ starting at $\mathfrak{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = b$. By the first property of Lemma 8.21, we can map each trace $\mathfrak{CT} \in \mathfrak{CTS}_b$ into a trace $\mathfrak{CT}^{\mathsf{cs}}$ using $\rightsquigarrow$ and starting at $\mathfrak{C}$, such that the configurations of the form $\mathfrak{C}$ of $\mathfrak{CT}^{\mathsf{cs}}$ are exactly the same as in $\mathfrak{CT}$ and $\Pr[\mathfrak{CT}^{\mathsf{cs}}] = \Pr[\mathfrak{CT}]$.

Let $\mathfrak{CTS}_b^{\mathsf{cs}}$ be the set of these traces $\mathfrak{CT}^{\mathsf{cs}}$. Let us show that $\mathfrak{CTS}_b^{\mathsf{cs}}$ is the set of complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathfrak{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = b$.

The list of events $\mathcal{E}$ in the last configuration of $\mathfrak{CT}^{\mathsf{cs}}$ is the same as in $\mathfrak{CT}$, so it satisfies $D(\mathcal{E}) = b$. By the second property of Lemma 8.21, since $\mathfrak{CT}$ is complete, $\mathfrak{CT}^{\mathsf{cs}}$ is also complete. Since the mapping from $\mathfrak{CT}$ to $\mathfrak{CT}^{\mathsf{cs}}$ is injective, we have $\Pr[\mathfrak{CTS}_b^{\mathsf{cs}}] = \Pr[\mathfrak{CTS}_b]$.

Moreover, if a configuration $\mathfrak{C}^{\mathsf{cs}}$ reduces by $\rightsquigarrow$ into another configuration, then the sum of the probabilities of all the possible reductions from $\mathfrak{C}^{\mathsf{cs}}$ is 1:

$$\sum_{\{\mathfrak{C}^{\mathsf{cs}\prime} | \mathfrak{C}^{\mathsf{cs}} \rightsquigarrow_{p(\mathfrak{C}^{\mathsf{cs}\prime})} \mathfrak{C}^{\mathsf{cs}\prime}\}} p(\mathfrak{C}^{\mathsf{cs}\prime}) = 1\,.$$

Indeed, the rules that define $\rightsquigarrow$ are mutually exclusive. If $\mathfrak{C}^{\mathsf{cs}}$ reduces by rule (CryptoVerif), then the property holds because it holds for the semantics of CryptoVerif. Otherwise, a single reduction is possible, and it has probability 1.

Using the same property for $\rightarrow$, the probability of all complete traces using $\rightarrow$ starting from $\mathfrak{C}$ is 1, so $\Pr[\mathfrak{CTS}_{\text{true}}] + \Pr[\mathfrak{CTS}_{\text{false}}] = 1$. So $\Pr[\mathfrak{CTS}_{\text{true}}^{\mathsf{cs}}] + \Pr[\mathfrak{CTS}_{\text{false}}^{\mathsf{cs}}] = 1$. Since the sum of the probabilities of all the possible reductions from each configuration by $\rightsquigarrow$ is 1, the probability of all complete traces using $\rightsquigarrow$ starting from $\mathfrak{C}$ is 1, so all these traces are in $\mathfrak{CTS}_{\text{true}}^{\mathsf{cs}}$ or $\mathfrak{CTS}_{\text{false}}^{\mathsf{cs}}$. Hence

all complete CryptoVerif traces using $\rightsquigarrow$ starting at $\mathfrak{C}$ and such that the list of events $\mathcal{E}$ in their last configuration satisfies $D(\mathcal{E}) = b$ are in $\mathfrak{CTG}_b^{\mathsf{cs}}$.

So $\Pr[\mathfrak{C}(\rightsquigarrow) : D] = \Pr[\mathfrak{CTG}_{\text{true}}^{\mathsf{cs}}] = \Pr[\mathfrak{CTG}_{\text{true}}] = \Pr[\mathfrak{C} : D]$. □

8.4.2 Relation between Intermediate and OCaml Semantics

Let us first give some preliminary definitions.

Definition 8.23 (Accessible trace, configuration) *An* accessible *trace $\mathcal{CT}$ (for the adversary program$_0$) is a trace beginning with the initial configuration $\mathcal{C}_0(Q_0, \mathit{program}_0)$ defined in Chapter 7.*

A configuration $\mathcal{C}$ is accessible *if there exists an accessible trace such that its last configuration is $\mathcal{C}$.*

Definition 8.24 (Concretization of $\mathcal{I}$ and $\mathcal{RI}$) *Let us define the sets of oracles $\mathcal{O}^\infty(\mathcal{I})$ and $\mathcal{O}^\infty(\mathcal{RI})$ represented by $\mathcal{I}$ and $\mathcal{RI}$ respectively:*

$$\begin{aligned}
\mathcal{O}^\infty(\mathcal{I}) &= \{O[b, \widetilde{a'}] \mid O\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{I}, a \leq b\} \cup \{O[\widetilde{a}] \mid O[\widetilde{a}] \in \mathcal{I}\} \\
\mathcal{O}^\infty(\mathcal{RI}) &= \{O[b, \widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[_, \widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}]), 1 \leq b\} \\
&\quad \cup \{O[\widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[\widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])\} \\
&\quad \cup \{O[b, \widetilde{a'}] \mid \mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{RI}, \\
&\qquad\qquad O[b, \widetilde{a'}] \in \mathsf{oracles}'(Q(\mathsf{role})[b, \widetilde{a'}]), a \leq b\}
\end{aligned}$$

The definition of $\mathcal{O}^\infty(\mathcal{I})$ and $\mathcal{O}^\infty(\mathcal{RI})$ ignores the replication bounds and allows the indices of oracles to go to infinity. Using unbounded indices is helpful in Properties 13 and 14 of Definition 8.32 below. By Assumption 5.4, when O is a first oracle of a role role under replication, O cannot be under replication in $Q(\mathsf{role})$. So the last component of $\mathcal{O}^\infty(\mathcal{RI})$ cannot contain oracles under replication.

Let us describe how the sets $\mathcal{I}$ and $\mathcal{RI}$ represent the contents of the set of callable processes $\mathcal{Q}$.

Definition 8.25 (Relation between $\mathcal{I}$, $\mathcal{RI}$ and $\mathcal{Q}$) *Let us define the sets of oracles $\mathcal{O}(\mathcal{I})$ and $\mathcal{O}(\mathcal{RI})$ represented by $\mathcal{I}$ and $\mathcal{RI}$ respectively:*

$$\begin{aligned}
\mathcal{O}(\mathcal{I}) &= \{O[b, \widetilde{a'}] \mid O\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{I}, a \leq b \leq N_O\} \cup \{O[\widetilde{a}] \mid O[\widetilde{a}] \in \mathcal{I}\} \\
\mathcal{O}(\mathcal{RI}) &= \{O[b, \widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, \\
&\qquad\qquad O[_, \widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}]), 1 \leq b \leq N_O\} \\
&\quad \cup \{O[\widetilde{a}] \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}, O[\widetilde{a}] \in \mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])\} \\
&\quad \cup \{O[b, \widetilde{a'}] \mid \mathsf{role}\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{RI}, \\
&\qquad\qquad O[b, \widetilde{a'}] \in \mathsf{oracles}'(Q(\mathsf{role})[b, \widetilde{a'}]), a \leq b \leq N_{\mathsf{role}}\}
\end{aligned}$$

We write $\mathcal{Q} \leftrightarrow \mathcal{RI}, \mathcal{I}$ when the following two properties hold:

- *$\mathcal{Q}$ consists of exactly one element $O[\widetilde{a}](x_1[\widetilde{a}] : T_1, \ldots, x_k[\widetilde{a}] : T_k) := P$ for each $O[\widetilde{a}]$ present in the set $\mathcal{O}(\mathcal{I}) \cup \mathcal{O}(\mathcal{RI})$. We denote by $\mathcal{Q}(O[\widetilde{a}])$ this element of $\mathcal{Q}$.*

- *If* $O\big[[a, +\infty[, \widetilde{a'}\big] \in \mathcal{I}$*, then there exist a process* Q *and an index* i *such that* i *does not occur in* $\mathsf{fv}(Q)$ *and for all* $b \in \{a, \ldots, N_O\}$*, we have* $\mathcal{Q}(O[b, \widetilde{a'}]) = Q\{b/i\}$*.*

In contrast to the sets we defined in Definition 8.24, the indices of oracles in $\mathcal{Q}$ are bounded by the replication bounds. So we redefine sets of oracles $\mathcal{O}(\mathcal{RI})$ and $\mathcal{O}(\mathcal{I})$ that correspond to $\mathcal{RI}$ and $\mathcal{I}$, but with indices bounded by N_O and N_{role} as appropriate. The sets $\mathcal{O}(\mathcal{RI})$ and $\mathcal{O}(\mathcal{I})$ are included in $\mathcal{O}^\infty(\mathcal{RI})$ and $\mathcal{O}^\infty(\mathcal{I})$, respectively. The set of processes $\mathcal{Q}$ corresponds to $\mathcal{RI}, \mathcal{I}$ when it contains exactly one definition for each oracle in $\mathcal{O}(\mathcal{I}) \cup \mathcal{O}(\mathcal{RI})$. Furthermore, in case an oracle is under replication, the corresponding elements of $\mathcal{Q}$ all have the same form; they differ only by the value of the replication index. We enforce this property in the last item of Definition 8.25.

Next, we define several sets of oracles and roles, which allow us to determine which oracles and roles are in which state (callable immediately, available later) in a simulator configuration.

Definition 8.26 (Oracle sets) *Let* $\mathcal{O}_{\mathsf{call}}(Th)$ *be the set of oracles* $O[\widetilde{a}]$ *not under replication that occur in* call *constructs in the thread* Th*, without entering tagged functions and closures.*

Let $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$ *be the set of oracles* $O[a, \widetilde{a}]$ *such that* O *is under replication,* $a > N_O$*, and* $\mathsf{call}(O[\_, \widetilde{a}])$ *occurs in the thread* Th*, without entering tagged functions and closures.*

Let $\mathcal{R}_{\mathsf{init\text{-}closure}}(Th)$ *be the set of roles* $\mathsf{role}[\widetilde{a}]$ *such that there exists* env *such that a closure* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *is present in the thread* Th*, and such that* $env(token)$ *is bound in its store to* true*.*

Let $\mathcal{R}_{\mathsf{init\text{-}function}}(Th)$ *be the set of roles* $\mathsf{role}[\widetilde{a}]$ *such that the initialization function* $program'(\mathsf{role}[\widetilde{a}])$ *is present in the thread* Th*.*

Let $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$*,* $\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS})$*, and* $\mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS})$ *be the unions of the corresponding sets for all threads of the configuration.*

We have already defined the set $\mathsf{returnoracles}(O[\widetilde{a}])$*, which is the set of oracles returned by* $O[\widetilde{a}]$*. Let us define* $\mathsf{returnoracles}'$ *such that:*

$$\begin{aligned}\mathsf{returnoracles}'(O[\widetilde{a}]) = {} & \{O'[\widetilde{a'}] \mid O'[\widetilde{a'}] \in \mathsf{returnoracles}(O[\widetilde{a}])\} \\ & \cup \{O'[b, \widetilde{a'}] \mid O'[\_, \widetilde{a'}] \in \mathsf{returnoracles}(O[\widetilde{a}]), 1 \leq b \leq N_{O'}\}\end{aligned}$$

Let $\mathcal{CS} = \mathcal{C}, \mathcal{RI}, \mathcal{I}$*. We denote the callable set of oracles:*

$$\mathsf{callable}(\mathcal{CS}) = \mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}^\infty(\mathcal{RI}) \cup \mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS}) \cup \mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS}))$$

Let $\mathsf{willbeavailable}(\mathcal{CS})$ *be the set of oracles that can eventually become available. This set is the least fixpoint of the function* f *defined by* $f(C) = C \cup \mathsf{returnoracles}'(C)$ *that contains the set* $\mathsf{returnoracles}'(\mathsf{callable}(\mathcal{CS}))$*.*

The definition of $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$ may be surprising, as it considers $O[a, \widetilde{a}]$ with a greater than the replication bound N_O. We have made this choice to guarantee that $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$ is always included in $\mathcal{O}^\infty(\mathcal{I})$: the indices up to N_O may have been consumed by calls already made to the oracle, while the indices greater

than N_O always remain, because we make at most N_O calls to this oracle by definition of N_O. This property is exploited in Property 14 of Definition 8.32 below.

The sets $\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS})$ and $\mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS})$ are sets of roles with their replication indices, which can be seen as a role set $\mathcal{RI}$. The set $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS}) \cup \mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS}))$ is the set of the first oracles of roles present in $\mathcal{R}_{\mathsf{init\text{-}closure}}(\mathcal{CS})$ and $\mathcal{R}_{\mathsf{init\text{-}function}}(\mathcal{CS})$.

The following definitions present functions used to relate OCaml and simulator threads.

Definition 8.27 (Replace initialization) *The function* replaceinitpm *replaces in its argument the pattern matchings corresponding to role initialization of the simulator by the OCaml module initialization: to be more precise,* $\mathsf{replaceinitpm}(Th)$ *replaces each occurrence of* $\mathsf{tagfunction}^{\mathsf{role}}\ pm'_{\mathsf{role}[\widetilde{a}]}$ *in* Th *with* $\mathsf{tagfunction}^{\mathsf{role}}\ pm_{\mu_{\mathsf{role}}}$ *and each occurrence of* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *in* Th *with* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm_{\mu_{\mathsf{role}}}]$.

This function transforms every occurrence of the tagged closures corresponding to role initialization in the simulator, which are added by the addthread construct, into the corresponding tagged closures in OCaml.

Definition 8.28 (Correct closure) *Suppose that* $\mathcal{Q} \leftrightarrow \mathcal{RI}, \mathcal{I}$ *for some* $\mathcal{RI}$, l_{tok} *is a function that maps each oracle* $O[\widetilde{a}]$ *to the location of its token, and* τ_{O} *is a function maps each oracle* $O[\_, \widetilde{a}]$ *to the tag* τ *of the corresponding closure. We define the set of closures that correspond to an oracle:*

- *for an oracle* $O[\widetilde{a}] \in \mathcal{I}$*:*

$$\begin{aligned}&\mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) =\\ &\quad \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{false}}(\mathcal{Q}(O[\widetilde{a}]))] \mid\\ &\quad\ env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}(O[\widetilde{a}])), env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\}\end{aligned}$$

- *for an oracle* $O[\widetilde{a}] \notin \mathcal{I}$*:*

$$\begin{aligned}&\mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) =\\ &\quad \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{false}}(Q)] \mid \textit{for any } Q, env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\}\end{aligned}$$

- *for an oracle* $O\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$ *with* $a' \leq N_O$*,*

$$\begin{aligned}&\mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) =\\ &\quad \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(\mathcal{Q}(O[a', \widetilde{a''}]))] \mid\\ &\quad\ \tau = \tau_{\mathsf{O}}(O[\_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}(O[a', \widetilde{a''}]))\}\end{aligned}$$

- *for an oracle* $O\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$ *with* $a' > N_O$*,*

$$\begin{aligned}&\mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) =\\ &\quad \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(Q)] \mid \tau = \tau_{\mathsf{O}}(O[\_, \widetilde{a''}]), \textit{for any } Q, env\}\end{aligned}$$

- *for an oracle* $O\big[[a', +\infty[, \widetilde{a''}\big] \notin \mathcal{I}$*:*

$$\mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \emptyset$$

The function correctclosure serves to map calls $\mathsf{call}(R)$ in the simulator configuration into their corresponding closures in the OCaml configuration: $\mathsf{call}(R)$ will be mapped below to an element of $\mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

In the case $O[\widetilde{a}] \in \mathcal{I}$, we map $\mathsf{call}(O[\widetilde{a}])$ into the closure that translates the process $\mathcal{Q}(O[\widetilde{a}])$.

The case $O[\widetilde{a}] \notin \mathcal{I}$ may be used when the oracle $O[\widetilde{a}]$ has been called but the thread still contains a call to this oracle. If the oracle is called again, the call will fail. The process $\mathcal{Q}(O[\widetilde{a}])$ is removed from $\mathcal{Q}$ after execution, so we do not know which process to translate to obtain the correct closure for $O[\widetilde{a}]$, that is why the correct closures for a call to an already called oracle can contain the translation of any process Q. This translation will fail and raise the exception Bad_Call regardless of the translated process Q.

Oracles under replication cannot disappear from $\mathcal{I}$ after having been added to it: when one calls the oracle $O[\_, \widetilde{a''}]$, we just increment the counter a' of the element $O\big[[a', +\infty[, \widetilde{a''}\big]$ present in $\mathcal{I}$. We need to distinguish whether the adversary has exhausted all the N_O calls available for this oracle or not. If there remains available calls, the process $\mathcal{Q}(O[a', \widetilde{a''}])$ is defined, and we require that $\mathsf{call}(O[\_, \widetilde{a''}])$ is mapped into a closure that translates this process. Otherwise, if all the calls are exhausted, $a' > N_O$, and $\mathcal{Q}(O[a', \widetilde{a''}])$ is not defined, but we know that the adversary will not call the oracle again, so $\mathsf{call}(O[\_, \widetilde{a''}])$ can be mapped to closures that translate any process.

The case $O\big[[a', +\infty[, \widetilde{a''}\big] \notin \mathcal{I}$ never happens: it would mean that the oracle $O[\_, \widetilde{a''}]$ can be called but there is no reference to it in the set $\mathcal{I}$.

Definition 8.29 (Replace call)

$\mathsf{replacecalls}(\langle env, pe, stack, store\rangle, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) =$
$\{\langle env', \sigma(pe), \sigma(stack), \sigma(store)\rangle \mid$ *if* pe *is a value* v *or an exceptional value* raise v*, then* env' *is any environment, else* $env' = \sigma(env)$*, where* σ *is a function that replaces, for each* R*, each occurrence of* $\mathsf{call}(R)$ *with an element of* $\mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\}$

The function replacecalls replaces in its argument each call $\mathsf{call}(O[\widetilde{a}])$ with a closure that corresponds to the oracle $O[\widetilde{a}]$, computed by correctclosure. It allows any environment when the current program or expression is a value or an exceptional value, because in these cases, the environment is not used.

Definition 8.30 (Token part of the store)

$$\begin{aligned}\mathsf{gettokens}(\mathcal{I}, \mathcal{O}, l_{\mathsf{tok}}) = {} & \{l_{\mathsf{tok}}(O[\widetilde{a}]) \mapsto \mathrm{true} \mid O[\widetilde{a}] \in \mathcal{O} \cap \mathcal{I}\} \\ & \cup \{l_{\mathsf{tok}}(O[\widetilde{a}]) \mapsto \mathrm{false} \mid O[\widetilde{a}] \in \mathcal{O} \setminus \mathcal{I}\}\end{aligned}$$

The function gettokens returns the part of the store corresponding to the tokens of the closures of oracles not under replication.

Definition 8.31 (Processes) *We use the following notations:*

P_{loop} *is the process from line 7 to line 20 in Figure 8.2.*

$Q_{loop} \stackrel{\text{def}}{=} O_{\text{loop}}[i'](s : T_{\mathcal{CS}}) := P_{loop}$.

$P_{return\text{-}loop}(\alpha) \stackrel{\text{def}}{=}$ if $b_{\alpha,r}[\,]$ then
let $r[\,] : T_{\mathcal{CS}} =$ loop $O_{loop}[\alpha+1](r'_{\alpha,r}[\,])$ in end else end
else $r[\,] \leftarrow r'_{\alpha,r}[\,]$; end .

$\mathcal{R}_{loop}(\alpha) \stackrel{\text{def}}{=} [((r'_{\alpha,r}[\,], b_{\alpha,r}[\,]), P_{return\text{-}loop}(\alpha), \text{end}), (x[\,], \text{return}(x[\,]), \text{end})]$.

Definition 8.32 (Relation between extended CryptoVerif configurations and OCaml traces) *Let $\mathfrak{C}^{\text{cs}}$ be an extended CryptoVerif configuration and $\mathcal{CT}$ be an accessible OCaml trace. We say that $\mathfrak{C}^{\text{cs}} \equiv \mathcal{CT}$ when there exists an injective function τ_{O} that maps oracles $O[\_, \widetilde{a}]$ such that $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$ for some a' to tags τ, such that the following properties are all true:*

1. $\mathfrak{C}^{\text{cs}} = E, P_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E}, steps, \mathcal{CS}$.
   $\mathcal{CS} = ([Th_1, \ldots, Th_n], globalstore, tj), \mathcal{RI}, \mathcal{I}$.
   For all $i \leq n$, $Th_i = \langle env_i, pe_i, stack_i, store_i\rangle$.

2. $\mathcal{C}$ *is the last configuration of* $\mathcal{CT}$.
   $\mathcal{C} = [Th'_1, \ldots, Th'_n], globalstore', tj, \mathcal{MI}, events$.
   For all $i \leq n$, $Th'_i = \langle env'_i, pe'_i, stack'_i, store'_i\rangle$.

3. $\mathcal{Q} = \{Q_{loop}\{a/i'\} \mid \alpha < a \leq N_{\text{rand+calls}}\} \cup \mathcal{Q}_0$ *and* $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}, \mathcal{I}$.

4. $\text{fv}(P_{loop}\{\alpha/i'\}) \cup \text{fv}(\mathcal{Q}) \cup \text{fv}(\mathcal{R}_{loop}(\alpha)) \subseteq \text{Dom}(E)$.

5. *For* $i \leq n$, *all store locations in* S_l *present in* Th_i *are in* $\text{Dom}(store_i)$.

6. *For each thread* $i \leq n$, *one of the following two cases occurs:*

   (a) $Th'_i = \text{replaceinitpm}(Th_i)$.
   $Th_i = \langle \emptyset, program_{\text{prim}}; ; program'(\text{role}_1[\widetilde{a_1}]); ; \ldots ; ; program'(\text{role}_l[\widetilde{a_l}]); ; program', [\,], \emptyset\rangle$.
   There is no closure, no tagged function tagfunction^t *pm, no* event, *and no* return *in program', except in* $program(\mu_{\text{role}})$ *in arguments of* addthread.

   (b) *The following properties hold:*

      i. *There exist* $store''_i$ *and an injective function* l_{tok} *that associates to each* $O[\widetilde{a}]$ *in* $\mathcal{O}_{\text{call}}(Th_i)$ *a store location that does not occur in* $\mathcal{CS}$ *such that*

$$\langle env'_i, pe'_i, stack'_i, store''_i\rangle \in \text{replacecalls}(\text{replaceinitpm}(Th_i), \mathcal{I}, E, \mathcal{Q}_0, l_{\text{tok}}, \tau_{\text{O}}),$$
$$store''_i \cup \text{gettokens}(\mathcal{I}, \mathcal{O}_{\text{call}}(Th_i), l_{\text{tok}}) \subseteq store'_i .$$

ii. *There exists an injective function* $l_{\mathsf{init\text{-}tok}}$ *that associates to each role* $\mathsf{role}[\widetilde{a}]$ *such that a closure* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *occurs in the thread* Th_i *for some* env *and* τ*, a store location such that for all closures* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *present in* Th_i*, we have* $l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}]) = env(token)$.
The locations $l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}])$ *and* $l_{\mathsf{tok}}(O[\widetilde{a'}])$ *are distinct for every role* $\mathsf{role}[\widetilde{a}]$ *and oracle* $O[a']$.
The locations $l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}])$ *occur only in* $\mathrm{Dom}(store_i)$ *and in* $env(token)$ *where* env *is the environment of a tagged closure* $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]$ *in* Th_i.

iii. *For each tagged closure* $\mathsf{tagfunction}^{t,\tau}[env, pm]$ *present in* Th_i*, the tag* t *is a role* role*,* $env_{\mathsf{prim}} \subseteq env$*, and there exist indices* $\widetilde{a}$ *such that* $pm = pm'_{\mathsf{role}[\widetilde{a}]}$.

iv. *There is no tagged function* $\mathsf{tagfunction}^t$ pm*, no* event*, and no* return *in* Th_i *except in* $program(\mu_{\mathsf{role}})$ *in arguments of* $\mathsf{addthread}$.

7. *For all locations* $l \in S_{priv}$*,* l *does not occur in* $Th_1, \ldots, Th_n$ *except in* $program(\mu_{\mathsf{role}})$ *in arguments of* $\mathsf{addthread}$.

8. $\forall l \in S_{priv}, globalstore(l) = initval_l$.

9. $globalstore(E, \mathcal{T}) \subseteq globalstore'$.

10. $\forall l \notin S_{priv}, globalstore(l) = globalstore'(l)$.

11. $\mathcal{MI} = \{(\mu_{\mathsf{role}}, \mathrm{false}) \mid \mathsf{role}[\widetilde{a}] \in \mathcal{RI}\} \cup \{(\mu_{\mathsf{role}}, \mathrm{true}) \mid \mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{RI}\}$.

12. $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$.

13. *The sets* $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$*,* $\mathcal{O}^\infty(\mathcal{RI})$*, and* $\mathsf{willbeavailable}(\mathcal{CS})$ *are pairwise disjoint.*

14. *The* $4n$ *sets of oracles* $\mathcal{O}_{\mathsf{call}}(Th_i)$*,* $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th_i)$*,* $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th_i))$*, and* $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th_i))$ *for* $i \leq n$ *are pairwise disjoint, and are all included in* $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$.

15. $|\mathcal{CT}| + steps \geq N_{\mathsf{steps}}$.

16. $\alpha \leq N_{\mathsf{rand}}(\mathcal{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT}) + 1$.

17. *If* $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$*, then* $a' \leq N_{\mathsf{calls}}(O, \tau_{\mathsf{O}}(O[\_, \widetilde{a}]), \mathcal{CT}) + 1$.

18. *If* $\mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{RI}$*, then* $a' \leq N_{\mathsf{exec}}(\mathsf{role}, \mathcal{CT}) + 1$.

The relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is our main tool to relate the CryptoVerif and OCaml systems. This relation holds only when the CryptoVerif adversary is evaluating the function *simulate* (line 7 of Figure 8.2), as shown by the form of the extended CryptoVerif configuration $\mathfrak{C}^{\mathsf{cs}}$ in Item 1. (The value α is the current value of the index i', that is, the number of iterations in the loop.) Items 1 and 2 also

ensure that there is the same number of threads in the simulator configuration $\mathcal{CS}$ and in the OCaml configuration $\mathcal{C}$.

Item 3 is an invariant on the CryptoVerif side: it relates the available oracles in $\mathcal{Q}$ to elements of the simulator configuration. This item ensures basically that when the simulator calls an oracle present in $\mathcal{I}$, it is also present in $\mathcal{Q}$, and the oracle call in the CryptoVerif adversary (line 13 of Figure 8.2) can proceed. Item 4 is an invariant of the CryptoVerif semantics: the environment contains bindings for every free variable present in the current configuration. Item 5 is an invariant of the simulator: each store location that occurs in a thread is present in the domain of the store. (When a location is created, it is immediately added to the store.)

Item 6 relates the threads of the simulator and of the OCaml semantics. A thread can be in one of the following two states. If it satisfies Item 6(a), the thread is a protocol thread that was not scheduled yet. The simulator and OCaml threads correspond by transforming the program $\mathit{program}'(\mathsf{role}[\widetilde{a}])$ present in the simulator into the program of the module corresponding to the role, $\mathit{program}(\mu_{\mathsf{role}})$. Otherwise, the thread satisfies Item 6(b). In this case, Item 6(b)i relates the contents of the simulator thread and the OCaml thread by replacing $\mathit{program}'(\mathsf{role}[\widetilde{a}])$ with $\mathit{program}(\mu_{\mathsf{role}})$ as above, and by replacing calls to oracles using call with a corresponding tagged closure. The tokens that determine whether oracles can be called are absent from the simulator: the value of these tokens is determined from $\mathcal{I}$ by the function $\mathsf{gettokens}$, and we require that they are present in the OCaml store with their correct value. Item 6(b)ii ensures that all instances of a closure of a given role initialization $\mathsf{role}[\widetilde{a}]$ share the same store location for their tokens. This ensures that a role initialization closure is not called twice. Item 6(b)ii also ensures that all locations used for the tokens of role initialization are not accessible elsewhere. Item 6(b)iii ensures that every tagged closure present in the simulator is a correct closure for the initialization of a role. Item 6(b)iv is an invariant of the simulator that ensures that the adversary does not have access to our OCaml instrumentation features.

Items 7 to 10 relate the values of the global store in the simulator and in the OCaml semantics. The public part of the global store is the same on both sides (Item 10). The private part (files and tables) is empty in the simulator, since this part is handled by CryptoVerif itself (Item 8) and cannot be accessed by the adversary (Item 7). We require that the private part of the OCaml global store corresponds to the CryptoVerif configuration (Item 9).

Item 11 relates the OCaml multiset of callable modules $\mathcal{MI}$ and the simulator set of callable roles $\mathcal{RI}$. Item 12 relates the OCaml and CryptoVerif events.

Items 13 and 14 present restrictions on sets of oracles. To understand how all these oracle sets interact, let us present the flow of an oracle not under replication $O[\widetilde{a}]$ in these sets.

1. Initially, if the oracle occurs at the beginning of the process, it is in $\mathcal{O}^{\infty}(\mathcal{RI})$; otherwise, it is in $\mathsf{willbeavailable}(\mathcal{CS})$.

2. For an oracle occurring at the beginning of a role, when the role containing it is instantiated using $\mathsf{addthread}$, the oracle moves from $\mathcal{O}^{\infty}(\mathcal{RI})$ to $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}function}}(\mathit{Th}))$. It is also added into $\mathcal{O}^{\infty}(\mathcal{I})$.

3. When the initialization function of the role is reduced into a closure, the oracle moves from $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th))$ to $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th))$.

4. When the initialization function of the role is called, the oracle moves from $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th))$ to $\mathcal{O}_{\mathsf{call}}(Th)$.

5. When the oracle itself is called, it is removed from $\mathcal{O}^\infty(\mathcal{I})$, and when the call to the oracle disappears from the thread, it is removed from $\mathcal{O}_{\mathsf{call}}(Th)$. The oracles made available after the call are removed from $\mathsf{willbeavailable}(\mathcal{CS})$ and added either to $\mathcal{O}^\infty(\mathcal{RI})$ if they start a role or to $\mathcal{O}^\infty(\mathcal{I})$ and $\mathcal{O}_{\mathsf{call}}(Th)$ if they do not start a role.

The case of an oracle under replication is fairly similar, using $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$ instead of $\mathcal{O}_{\mathsf{call}}(Th)$. Items 13 and 14 ensure that an oracle cannot be simultaneously in two different sets. These properties allow us to prove that the injections of Items 6(b)i and 6(b)ii are kept. (We use $\mathcal{O}^\infty$ rather than $\mathcal{O}$ to make sure that we cannot have simultaneously $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$ with $a' > N_O$ and $O[b, \widetilde{a}] \in \mathsf{willbeavailable}(\mathcal{CS})$ for all b. Indeed, $\mathcal{O}(\{O\big[[a', +\infty[, \widetilde{a}\big]\})$ is empty when $a' > N_O$, so this situation would not be prevented by Item 13 if it used $\mathcal{O}$. It is prevented using $\mathcal{O}^\infty$.)

Items 15 to 18 ensure that we never reach the limits on the number of simulator steps N_{steps} (Item 15), the number of calls to the oracles (Item 16 for the oracle O_{loop} and Item 17 for the other oracles), and the number of calls to roles (Item 18), by making sure that the number of calls on the CryptoVerif side is at most the number of calls on the OCaml side. The number of calls made to oracle $O[\_, \widetilde{a}]$ in CryptoVerif, $a' - 1$ such that $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$, may be less than the number of calls to that oracle in the OCaml trace, $N_{\mathsf{calls}}(O, \tau_{\mathsf{O}}(O[\_, \widetilde{a}]), \mathcal{CT})$, because failed calls are not counted on the CryptoVerif side.

The next two lemmas show that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved during execution. Lemma 8.33 shows that it holds at the beginning, as soon as the simulator reaches line 7 of Figure 8.2.

Lemma 8.33 *There exists a trace $\mathfrak{C}_0(Q_0, \mathit{program}_0) \rightsquigarrow^* \mathfrak{C}^{\mathsf{cs}}$ where $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}_0$ and $\mathcal{CT}_0 = \mathcal{C}_0(Q_0, \mathit{program}_0)$.*

Lemma 8.34 shows that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved. More precisely, the relation does not hold at all steps (in particular because it holds only when the CryptoVerif adversary is executing *simulate*), but if it holds at some point, we can continue execution so that either it holds again at a later point, or execution ends with matching events.

Lemma 8.34 *Let $\mathfrak{C}^{\mathsf{cs}}$ such that there exists an accessible trace $\mathcal{CT}$ satisfying $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.*

- *Either there exist n configurations $\mathfrak{C}^{\mathsf{cs}}_1, \ldots, \mathfrak{C}^{\mathsf{cs}}_n$ and n traces $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow^+_{p_1} \mathfrak{C}^{\mathsf{cs}}_1$, $\ldots, \mathfrak{C}^{\mathsf{cs}} \rightsquigarrow^+_{p_n} \mathfrak{C}^{\mathsf{cs}}_n$ such that none of these traces is a prefix of another, $\sum_{i \leq n} p_i = 1$, and for each accessible trace $\mathcal{CT}$ such that $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$, there exist n pairwise disjoint trace sets $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ such that all traces in these sets are extensions of $\mathcal{CT}$, none of these traces is a prefix of another, $\Pr[\mathcal{CTS}_i] = p_i \cdot \Pr[\mathcal{CT}]$, and for each trace $\mathcal{CT}' \in \mathcal{CTS}_i$, we have $\mathfrak{C}^{\mathsf{cs}}_i \equiv \mathcal{CT}'$.*

- *Or for each accessible trace* $\mathcal{CT}$ *such that* $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$*, the last configuration* $\mathcal{C}$ *of* $\mathcal{CT}$ *cannot reduce,* $\mathfrak{C}^{\mathsf{cs}} \rightarrow^{+} \mathfrak{C}_1^{\mathsf{cs}}$*, the configuration* $\mathfrak{C}_1^{\mathsf{cs}}$ *cannot reduce, and the event list* $\mathcal{E}$ *of* $\mathfrak{C}_1^{\mathsf{cs}}$ *and the event list events of* $\mathcal{C}$ *satisfy* $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$*.*

We prove these lemmas in Appendix B. Let us present a proof sketch of Lemma 8.34.

Proof sketch Let us take an extended CryptoVerif configuration $\mathfrak{C}^{\mathsf{cs}}$ and an OCaml trace $\mathcal{CT}$ such that $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. Let $\mathcal{C}$ be the last configuration of $\mathcal{CT}$. Let $\mathcal{CS}$ be the configuration of the simulator in $\mathfrak{C}^{\mathsf{cs}}$ and Th be the current thread of $\mathcal{CS}$.

Case 1: the current thread of $\mathcal{CS}$ verifies Item 6(a), we run the initialization of the module. The programs of the current threads of $\mathcal{CS}$ and $\mathcal{C}$ are the same except that $program'(\mathsf{role}[\widetilde{a}])$ present in $\mathcal{CS}$ are transformed into $program(\mu_{\mathsf{role}})$. We show that after having reduced the initialization of the primitives and the initialization of the roles in both sides, the current threads verify Item 6(b). The oracles in $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}function}}(Th))$ that correspond to the roles implemented in this initialization are moved to $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th))$. We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

Case 2: the current thread of $\mathcal{CS}$ verifies Item 6(b). We distinguish cases on the form of the simulator configuration $\mathcal{CS}$.

Let us first look at the cases in which the configuration $\mathcal{CS}$ does not reduce. We use the rule (Leave Simulator), thus finishing the evaluation of the function *simulate*.

- If the current expression of $\mathcal{CS}$ is $\mathsf{call}(O_j[\widetilde{a}])\ v$, then the result of *simulate* is such that $o = o_j$, so the CryptoVerif adversary of Figure 8.2 calls the oracle O_j at line 13 in the branch $o = o_j$, ends one iteration of O_{loop}, and starts the next iteration until it reaches line 7. We use Lemma 8.15 and we exploit the definition of $simulate'_{O_j}$ and $simulate''_{O_j}$ to prove that the OCaml configuration reduces similarly, by calling the OCaml function generated for oracle O_j. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}^{\infty}(\mathcal{I})$, and from $\mathcal{O}_{\mathsf{call}}(Th)$ if all occurrences of $\mathsf{call}(O_j[\widetilde{a}])$ have disappeared. The newly available oracles, added to sets $\mathcal{O}^{\infty}(\mathcal{RI})$ or $\mathcal{O}_{\mathsf{call}}(Th)$ and $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$, are removed from the set $\mathsf{willbeavailable}(\mathcal{CS})$. We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

- If the current expression of $\mathcal{CS}$ is $\mathsf{random}\ ()$, then the result of *simulate* is such that $o = o_R$, so the CryptoVerif adversary of Figure 8.2 samples a random boolean at line 19, ends one iteration of O_{loop}, and starts the next iteration until it reaches line 7. The current expression of $\mathcal{CS}$ is replaced with true with probability 1/2 and false with probability 1/2. The OCaml configuration reduces similarly: it samples a random boolean by using the rule (Random), and the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

- Otherwise, the configuration $\mathcal{CS}$ cannot reduce, and the corresponding configuration $\mathcal{C}$ cannot reduce either. The result of *simulate* is such that $o = o_S$, so the CryptoVerif adversary of Figure 8.2 ends the current iteration of O_{loop} at line 9, and ends the loop at line 4, so it also stops.

The events in the final CryptoVerif and OCaml configurations match, so the second case of the lemma holds.

If the current expression of $\mathcal{CS}$ is $\mathsf{addthread}(\mathit{program})$, a new thread is created on both sides. If *program* is a protocol program, then this new thread satisfies Item 6(a) by definition of addthread in OCaml and in the simulator and by definition of replaceinitpm. The roles added in this new thread Th are removed from $\mathcal{RI}$ and the corresponding oracles are added to $\mathcal{O}^{\infty}(\mathcal{R}_{\mathsf{init\text{-}function}}(Th))$ and to $\mathcal{I}$. Otherwise, the new thread satisfies Item 6(b). We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

If the current expression of $\mathcal{CS}$ is $\mathsf{call}(O_j[\widetilde{a}])\ v$ and $\mathcal{CS}$ reduces by rule (FailedCall1) or (FailedCall2), then the simulator raises Bad_Call, and the corresponding tagged function in OCaml also raises Bad_Call (because the tokens in OCaml correspond to $\mathcal{I}$ in the simulator by Item 6(b)i). We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

If the current expression of $\mathcal{CS}$ is $\mathsf{tagfunction}^{\mathsf{role},\tau}[env, pm'_{\mathsf{role}[\widetilde{a}]}]\ ()$, then we execute the initialization function of role role. We remove this role from the set $\mathcal{R}_{\mathsf{init\text{-}closure}}(Th)$, and add the corresponding oracles to $\mathcal{O}_{\mathsf{call}}(Th)$ and $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th)$. We prove that the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$ is preserved.

The other cases are straightforward since the simulator mimics the OCaml semantics. They all preserve the relation $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. □

From Lemmas 8.33 and 8.34, we can prove the following proposition, by extending the traces using Lemma 8.34 until we get complete traces.

Proposition 8.35 *Let $\mathfrak{CT}_1, \ldots, \mathfrak{CT}_n$ be complete CryptoVerif traces starting at $\mathfrak{C}_0(Q_0, \mathit{program}_0)$.*

Then there exist disjoint sets of complete OCaml traces $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ all starting at $\mathcal{C}_0(Q_0, \mathit{program}_0)$ such that for all $i \leq n$, $\Pr[\mathfrak{CT}_i] = \Pr[\mathcal{CTS}_i]$, and if $\mathfrak{C}$ is the last configuration of $\mathfrak{CT}_i$ and $\mathcal{C}$ is the last configuration of a trace in $\mathcal{CTS}_i$, then the event list $\mathcal{E}$ of $\mathfrak{C}$ and the event list events of $\mathcal{C}$ satisfy $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$.

To prove this result, we begin by defining the relation $\equiv_{\mathsf{t}}$ between lists of CryptoVerif traces and lists of OCaml trace sets.

Definition 8.36 *The relation $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ is verified when the following properties hold:*

1. *All traces $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$ start at $\mathfrak{C}_0(Q_0, \mathit{program}_0)$, and none of these traces is a prefix of another of these traces.*
2. *The trace sets $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ are pairwise disjoint, all traces in these sets start at $\mathcal{C}_0(Q_0, \mathit{program}_0)$, and none of these traces is a prefix of another of these traces.*
3. *$\forall i \leq n, \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] = \Pr[\mathcal{CTS}_i]$.*
4. *$\sum_{i \leq n} \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] = 1$.*
5. *For each trace $\mathfrak{CT}_i^{\mathsf{cs}}$, $i \leq n$,*

(a) *either* $\mathfrak{CT}_i^{\mathsf{cs}}$ *is complete, every trace* $\mathcal{CT} \in \mathcal{CTS}_i$ *is complete, and the event list* $\mathcal{E}$ *of the last configuration of* $\mathfrak{CT}_i^{\mathsf{cs}}$ *and the event list events of the last configuration of* $\mathcal{CT}$ *verify* $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$,

(b) *or for every trace* $\mathcal{CT} \in \mathcal{CTS}_i$, *the last configuration* $\mathfrak{C}^{\mathsf{cs}}$ *of* $\mathfrak{CT}_i^{\mathsf{cs}}$ *verifies* $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

Items 1 and 4 show that whatever CryptoVerif complete trace $\mathfrak{CT}^{\mathsf{cs}}$ beginning at $\mathfrak{C}_0(Q_0, program_0)$, there is an unique trace $\mathfrak{CT}_i^{\mathsf{cs}}$ such that $\mathfrak{CT}_i^{\mathsf{cs}}$ is a prefix of $\mathfrak{CT}^{\mathsf{cs}}$. Item 3 and 5 show how the OCaml trace sets are related to the CryptoVerif traces. The idea of the following lemmas is to first find, using Lemma 8.33, an OCaml trace set $\mathcal{CTS}$ such that $\mathfrak{C}_0(Q_0, program_0) \equiv_{\mathsf{t}} \mathcal{CTS}$, and then, while there is a non complete trace in the CryptoVerif traces list, reduce this trace using Lemma 8.34 to obtain a new list of CryptoVerif traces and OCaml trace sets that correspond using $\equiv_{\mathsf{t}}$.

The next lemma applies to any traces, so in particular to OCaml traces and CryptoVerif traces.

Lemma 8.37 *Let* $\mathcal{CT}_1, \ldots, \mathcal{CT}_n$ *be traces such that none of these traces is a prefix of another of these traces. If* $\mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$ *are extensions of* $\mathcal{CT}_n$ *such that none of these traces is a prefix of another, then none of the traces* $\mathcal{CT}_1, \ldots, \mathcal{CT}_{n-1}, \mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$ *is a prefix of another of these traces.*

In particular, this is true when, for all $i \leq n'$, $\mathcal{CT}_i{}''$ *is the concatenation of* $\mathcal{CT}_n$ *and* $\mathcal{CT}_i{}'$ *where* $\mathcal{CT}_1{}', \ldots, \mathcal{CT}_{n'}{}'$ *are traces that start at the last configuration of* $\mathcal{CT}_n$ *such that none of these traces is a prefix of another of these traces.*

Proof Let us prove the first point. Consider two traces among $\mathcal{CT}_1, \ldots, \mathcal{CT}_{n-1}$, $\mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$. If they are both among $\mathcal{CT}_1, \ldots, \mathcal{CT}_{n-1}$, they are not prefix of one another by hypothesis. If they are both among $\mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$, they are also not prefix of one another by hypothesis. Now consider $\mathcal{CT}_i$ with $i \leq n-1$ and $\mathcal{CT}_j{}''$ with $j \leq n'$. If $\mathcal{CT}_i$ was a prefix of $\mathcal{CT}_j{}''$, then either its length is less or equal to the length of $\mathcal{CT}_n$, so $\mathcal{CT}_i$ would be a prefix of $\mathcal{CT}_n$, which is impossible by hypothesis, or its length is greater than the length of $\mathcal{CT}_n$, so $\mathcal{CT}_i$ would be an extension of $\mathcal{CT}_n$, that is, $\mathcal{CT}_n$ would be a prefix of $\mathcal{CT}_i$, which is also impossible by hypothesis. If $\mathcal{CT}_j{}''$ was a prefix of $\mathcal{CT}_i$, then a fortiori $\mathcal{CT}_n$ would be a prefix of $\mathcal{CT}_i$, which is impossible by hypothesis. Hence, none of the traces $\mathcal{CT}_1, \ldots, \mathcal{CT}_{n-1}, \mathcal{CT}_1{}'', \ldots, \mathcal{CT}_{n'}{}''$ is a prefix of another of these traces.

To show the second point, if $\mathcal{CT}_i{}''$ was a prefix of $\mathcal{CT}_j{}''$, then $\mathcal{CT}_i{}'$ would be a prefix of $\mathcal{CT}_j{}'$, which is a contradiction. So we can apply the first point in this case. □

The following lemma shows the induction step.

Lemma 8.38 *Suppose that* $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$. *Either all traces* $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$ *are complete, or there exist* $\mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime}$ *and* $\mathcal{CTS}'_1, \ldots, \mathcal{CTS}'_{n'}$ *such that there are strictly more reduction steps in traces* $\mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime}$ *than in traces* $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$ *and* $\mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}'_1, \ldots, \mathcal{CTS}'_{n'}$.

Proof Suppose that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ and there is a trace $\mathfrak{CT}_i^{\mathsf{cs}}$ that is not complete. We can renumber the traces so that the last trace $\mathfrak{CT}_n^{\mathsf{cs}}$ is not complete.

By Property 5(b), the last configuration $\mathfrak{C}^{\mathsf{cs}}$ of the trace $\mathfrak{CT}_n^{\mathsf{cs}}$ and all traces $\mathcal{CT} \in \mathcal{CTS}_n$ verify $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. By Property 2, $\mathcal{CT}$ is a trace beginning at $\mathcal{C}_0(Q_0, \mathit{program}_0)$. Let us denote $\mathcal{CTS}_n = \{\mathcal{CT}_1, \ldots, \mathcal{CT}_m\}$. We can then apply Lemma 8.34 to $\mathfrak{C}^{\mathsf{cs}}$.

- Either there exist n' configurations $\mathfrak{C}_1^{\mathsf{cs}}, \ldots, \mathfrak{C}_{n'}^{\mathsf{cs}}$, n' traces $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow_{p_1}^{+} \mathfrak{C}_1^{\mathsf{cs}}$, $\ldots, \mathfrak{C}^{\mathsf{cs}} \rightsquigarrow_{p_{n'}}^{+} \mathfrak{C}_{n'}^{\mathsf{cs}}$ such that none of these traces is a prefix of another, $\sum_{i \leq n} p_i = 1$, and for each trace $\mathcal{CT}_j, j \leq m$, there exist n' pairwise disjoint trace sets $\mathcal{CTS}_{j,1}, \ldots, \mathcal{CTS}_{j,n'}$ such that all traces in these sets are extensions of $\mathcal{CT}_j$, none of these traces is a prefix of another, and for each trace $\mathcal{CT} \in \mathcal{CTS}_{j,i}, \mathfrak{C}_i^{\mathsf{cs}\prime} \equiv \mathcal{CT}$ and $\Pr[\mathcal{CTS}_{j,i}] = p_i \cdot \Pr[\mathcal{CT}_j]$. Let us denote $\mathcal{CTS}'_i \stackrel{\text{def}}{=} \bigcup_{j \leq m} \mathcal{CTS}_{j,i}$. Let us also denote $\mathfrak{C}_i^{\mathsf{cs}\prime}$ the extension of the trace $\mathfrak{C}^{\mathsf{cs}}$ until $\mathfrak{C}_i^{\mathsf{cs}}$, for $i \leq n'$. There is at least one new reduction step, so there are more reduction steps in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime}$ than in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$. Let us prove that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_{n-1}, \mathcal{CTS}'_1, \ldots, \mathcal{CTS}'_{n'}$. All traces $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime}$ start at $\mathfrak{C}_0(Q_0, \mathit{program}_0)$ and by Lemma 8.37, none of these traces is a prefix of another of these traces, so Property 1 is verified. Similarly, by applying Lemma 8.37 to each trace $\mathcal{CT}_j$ for $j \leq m$, Property 2 is verified. By Property 3 on the initial traces, $\forall i \leq n, \Pr[\mathfrak{C}_i^{\mathsf{cs}}] = \Pr[\mathcal{CTS}_i]$. We have that $\Pr[\mathcal{CTS}'_i] = \sum_{j \leq m} \Pr[\mathcal{CTS}_{j,i}]$ because all the sets $\mathcal{CTS}_{j,i}$ are disjoint. So,

$$\Pr[\mathcal{CTS}'_i] = \sum_{j \leq m} p_i \cdot \Pr[\mathcal{CT}_j] = p_i \cdot \Pr[\mathcal{CTS}_n]\,, \text{ so}$$
$$\Pr[\mathfrak{CT}_i^{\mathsf{cs}\prime}] = p_i \cdot \Pr[\mathfrak{CT}_n^{\mathsf{cs}}] = \Pr[\mathcal{CTS}'_i]\,,$$

Property 3 is verified. By Property 4 on the initial traces, $\sum_{i \leq n} \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] = 1$. We have that

$$\sum_{i \leq n'} \Pr[\mathfrak{CT}_i^{\mathsf{cs}\prime}] = \sum_{i \leq n'} p_i \cdot \Pr[\mathfrak{CT}_n^{\mathsf{cs}}] = \Pr[\mathfrak{CT}_n^{\mathsf{cs}}]\,, \text{ so}$$
$$\sum_{i \leq n-1} \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] + \sum_{i \leq n'} \Pr[\mathfrak{CT}_i^{\mathsf{cs}\prime}] = \sum_{i \leq n} \Pr[\mathfrak{CT}_i^{\mathsf{cs}}] = 1\,.$$

So Property 4 is verified. We inherit Property 5 for the $n-1$ first elements. For all $i \leq n'$, for all traces $\mathcal{CT} \in \mathcal{CTS}'_i$, we have $\mathfrak{C}_i^{\mathsf{cs}\prime} \equiv \mathcal{CT}$, and $\mathfrak{C}_i^{\mathsf{cs}\prime}$ is the last configuration of $\mathfrak{CT}_i^{\mathsf{cs}\prime}$. So Property 5(b) is verified for all the new elements. Hence Property 5 is verified. Therefore, $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_1^{\mathsf{cs}\prime}, \ldots, \mathfrak{CT}_{n'}^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_{n-1}, \mathcal{CTS}'_1, \ldots, \mathcal{CTS}'_{n'}$.

- Otherwise, each trace $\mathcal{CT} \in \mathcal{CTS}_n$ is complete, $\mathfrak{C}^{\mathsf{cs}} \rightarrow^* \mathfrak{C}_1^{\mathsf{cs}}$, $\mathfrak{C}_1^{\mathsf{cs}}$ cannot reduce, and the event list $\mathcal{E}$ of $\mathfrak{C}_1^{\mathsf{cs}}$ and the event list *events* of the last configuration of $\mathcal{CT}$ satisfy $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$. Let $\mathfrak{CT}_n^{\mathsf{cs}\prime}$ be the extension of the trace $\mathfrak{CT}_n^{\mathsf{cs}}$ until $\mathfrak{C}_1^{\mathsf{cs}}$. The trace $\mathfrak{CT}_n^{\mathsf{cs}\prime}$ contains more steps than $\mathfrak{CT}_n^{\mathsf{cs}}$,

so there are more reduction steps in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_n^{\mathsf{cs}\prime}$ than in $\mathfrak{CT}_1^{\mathsf{cs}}$, $\ldots, \mathfrak{CT}_n^{\mathsf{cs}}$. Let us prove that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_n^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$. By Lemma 8.37, no traces in $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_n^{\mathsf{cs}\prime}$ are prefixes of one another, so Property 1 is verified. Property 2 is inherited. We have that $\Pr[\mathfrak{CT}_n^{\mathsf{cs}\prime}] = \Pr[\mathfrak{CT}_n^{\mathsf{cs}}]$, so Properties 3 and 4 are verified. The trace $\mathfrak{CT}_n^{\mathsf{cs}\prime}$ is complete, every trace $\mathcal{CT} \in \mathcal{CTS}_n$ is complete, and the event list *events* of the last configuration of traces in $\mathcal{CTS}_n$ and the event list $\mathcal{E}$ of $\mathfrak{C}_1^{\mathsf{cs}}$ verify $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$, so Property 5(a) holds for the last elements. Other elements inherit Property 5, so Property 5 holds. Therefore, $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_{n-1}^{\mathsf{cs}}, \mathfrak{CT}_n^{\mathsf{cs}\prime} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$. □

Proof (of Proposition 8.35) By Lemma 8.33, we have a trace $\mathfrak{CT}_0 = \mathfrak{C}_0(Q_0, \mathit{program}_0) \rightsquigarrow^* \mathfrak{C}^{\mathsf{cs}}$ where $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}_0$ and $\mathcal{CT}_0 = \mathcal{C}_0(Q_0, \mathit{program}_0)$. We prove easily that $\mathfrak{CT}_0 \equiv_{\mathsf{t}} \{\mathcal{CT}_0\}$. The number of steps in complete traces from configuration $\mathfrak{C}_0(Q_0, \mathit{program}_0)$ is finite. Let us consider traces such that $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}} \equiv_{\mathsf{t}} \mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ with the maximum number of reduction steps. By Lemma 8.38, the traces $\mathfrak{CT}_1^{\mathsf{cs}}, \ldots, \mathfrak{CT}_n^{\mathsf{cs}}$ are complete. (Otherwise, we could extend them.) Since the sum of their probabilities is 1, these are all complete traces starting at $\mathfrak{C}_0(Q_0, \mathit{program}_0)$. The proposition follows. □

As an immediate consequence of this proposition, we obtain:

Proposition 8.39 $\Pr[\mathfrak{C}_0(Q_0, \mathit{program}_0)(\rightsquigarrow) : D] = \Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D]$.

8.4.3 Security Result

By combining Propositions 8.22 and 8.39, we obtain the following theorem:

Theorem 8.40 (Security result)

$$\Pr[\mathfrak{C}_0(Q_0, \mathit{program}_0) : D] = \Pr[\mathcal{C}_0(Q_0, \mathit{program}_0) : D] \,.$$

In other words, the adversary $\mathit{program}_0$ against our generated OCaml modules has the same probability of breaking the security property as the adversary $Q_{\mathrm{adv}}(Q_0, \mathit{program}_0)$ against the CryptoVerif process.

CryptoVerif bounds the probability that an adversary Q breaks the security property D, that is, it finds a probability p that depends on the adversary such that, for all CryptoVerif adversaries Q for Q_0,

$$\Pr[\mathfrak{C}_{\mathrm{i}}(Q_0 \mid Q) : D] \leq p \,.$$

The adversaries $Q_{\mathrm{adv}}(Q_0, \mathit{program}_0)$ are CryptoVerif adversaries for Q_0, so for all OCaml programs *program* that obey our assumptions,

$$\Pr[\mathcal{C}_0(Q_0, \mathit{program}) : D] = \Pr[\mathfrak{C}_0(Q_0, \mathit{program}) : D] \leq p$$

Hence, all considered OCaml adversaries *program* can break the security property D with probability at most p.

The probability bound p returned by CryptoVerif is a function that depends on many parameters, expressed on the CryptoVerif protocol specification. Let us relate these parameters to the OCaml implementation. These parameters are as follows:

- The maximum number of times the various oracles and roles have been called, N_O and N_{role}. As shown by our proof and by Definition 8.16, N_O can be set to the maximum number of calls to the same closure representing oracle O in any trace of the OCaml program, and N_{role} can be set to the maximum number of instantiations of the role role in any trace of this program.

- The size of the CryptoVerif types T. The corresponding OCaml type $\mathbb{G}_T(T)$ is fixed by the annotations of the CryptoVerif specification. The size of T can be set to the size of $\mathbb{G}_T(T)$. Similarly, the size of the CryptoVerif values a (used when their type T has unbounded size) can be set to the size of the corresponding OCaml value $\mathbb{G}_{\mathsf{val}T}(a)$.

- The execution time of the cryptographic primitives and of various CryptoVerif constructs. This time can be set to the execution time of the corresponding OCaml implementation.

- The execution time of the adversary. Our proof shows that the function *simulate* executes at most as many reduction steps as the OCaml adversary. However, the CryptoVerif adversary shown in Figure 8.2 also includes additional steps and conversions between the OCaml semantic configuration and its CryptoVerif bitstring representation. By using the contents of the OCaml memory as bitstring representation of the semantic configuration in CryptoVerif, we can obtain an efficient implementation of the CryptoVerif adversary that does not take significantly more time than the OCaml adversary.

From the probability bound given by CryptoVerif, we can then obtain a bound on the probability of breaking the security properties in the generated OCaml implementation of the protocol.

As detailed in Chapter 4, CryptoVerif shows that our model of the SSH Transport Layer Protocol guarantees the authentication of the server to the client and the secrecy of the session keys. By Theorem 8.40, our generated implementation of this protocol satisfies the same properties, provided assumptions A1 to A6 hold.

Conclusion

We have proved that our compiler preserves security. Therefore, by using CryptoVerif, we can prove the desired security properties on the protocol specification, and then by using our compiler, we get a runnable implementation of the protocol, which satisfies the same security properties as the specification. Making such a proof is also useful because it clarifies the assumptions needed to ensure that the implementation is secure (Assumptions A1 to A6 in our case). The proof technique we presented, which is simulating any adversary by a CryptoVerif process, is also useful to show that any Turing machine can be encoded as a CryptoVerif adversary, which is important for the validity of the verification by CryptoVerif.

Conclusion

To summarize, we presented our compiler that translates annotated CryptoVerif specifications into OCaml implementations. We presented how we model the SSH Transport Layer Protocol, and the extensions we have done to CryptoVerif to be able to prove the authentication of the server to the client and the secrecy of the generated keys. We described these proofs, and we generated with the help of our compiler an actual implementation that is able to interact with OpenSSH. We then presented the proof of correctness of our compiler under some reasonable assumptions.

Let us state again our goal: obtain secure implementations of cryptographic protocols. We believe that, to obtain a secure implementation of a protocol, we first need to be sure that the protocol is indeed secure, because it is impossible to fix an insecure implementation of a protocol when the protocol is flawed, without changing the protocol. That is why we would like to highlight our methodology of first writing a protocol specification, prove it, and only then implement the protocol.

Future Work

We can identify two principal future work categories. We can improve our compiler so we get a better security result, or we can improve the proof, so that we have better confidence in the correctness of the generated implementation.

Compiler Improvements

To prove the security of the complete implementation, we need to be sure that the assumptions we made in the proof are respected. Most of these assumptions can be proved with static analysis. For example, one can test whether our network code is well-typed (Assumption A4), does not mutate strings received by our generated implementation (Assumption A5), and does not fork when a role is executed (Assumption A6). It would be interesting to automate this verification. An interesting, but more difficult endeavor, would be to prove that cryptographic primitives are correct with respect to the specification (Assumption A1). There are two kinds of specifications one can give to a primitive. One can give syntactic assumptions, for example that the decryption of the ciphertext under the correct key yields the plaintext. One can check that these assumptions are correct by static analysis on the implementation of the primitive. The second kind of assumptions is security assumptions, like saying that the encryption is IND-CPA.

To prove these, one can extract a model of the primitive, and then prove the correction of this model. Asymmetric encryption primitives can be modeled using the work by Courant et al. [26]. More generally, the CertiCrypt tool [7] is also able to model all kinds of primitives.

Our generated implementations do not include countermeasures against side-channel attacks. It would be interesting to add such countermeasures, or even to have tools to detect certain side-channel attacks or prove their absence. This is long-term future work.

Our implementation of SSH is usable, but the performance is not as good as one would want. Some oracles will be called many times, and so it is important that these oracles are efficient. We could try to change our compiler to optimize more the code of the CryptoVerif constructs. This work would also need a new proof of correctness of the translation, similar to what we presented in Section 8.2. One could also try optimizing the cryptographic primitives themselves.

Proof improvements

We have done the proof by hand. Formalizing it using a proof assistant (e.g., Coq) would be interesting future work.

CryptoVerif terms are in fact more complicated than what we described in Chapter 1. Terms can contain tests and find**s**, because CryptoVerif can use this to its advantage in order to devise proofs of protocols. Our proof does not use the complete CryptoVerif terms syntax, nor our letfun syntax improvement, and it would be interesting to extend our proof to support these.

CryptoVerif Improvements

We would also like to extend CryptoVerif to support states. Many protocols are stateful, and this would permit the tool to analyze many more protocols. In particular, we could then analyze the counter mode encryption and prove the correctness of the complete SSH protocol, when one uses this encryption scheme.

Bibliography

[1] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *First IFIP International Conference on Theoretical Computer Science*, volume 1872 of *LNCS*, pages 3–22. Springer, 2000.

[2] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In *CCS'11*, pages 331–340, New York, 2011. ACM.

[3] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational verification of C protocol implementations by symbolic execution. In *CCS'12*, pages 712–723, New York, 2012. ACM.

[4] Martin R. Albrecht, Kenny G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *IEEE Symposium on Security and Privacy*, pages 16–26. IEEE, 2009.

[5] Matteo Avalle, Alfredo Pironti, Riccardo Sisto, and Davide Pozza. The JavaSPI framework for security protocol implementation. In *ARES'11*, pages 746–751. IEEE, 2011.

[6] Michael Backes, Dennis Hofheinz, and Dominique Unruh. CoSP: A general framework for computational soundness proofs. In *CCS'09*, pages 66–78. ACM, 2009.

[7] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella. Formal certification of code-based cryptographic proofs. In *POPL'09*, pages 90–101. ACM, 2009.

[8] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *FOCS'97*, pages 394–403. IEEE, 1997.

[9] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Authenticated encryption in SSH: Provably fixing the SSH binary packet protocol. In *CCS'02*, pages 1–11. ACM, 2002.

[10] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. The secure shell (SSH) transport layer encryption modes. `http://www.ietf.org/rfc/rfc4344.txt`, 2006.

[11] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andy Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM TOPLAS*, 33(2), 2011.

[12] Karthikeyan Bhargavan, Ricardo Corin, Cédric Fournet, and Eugen Zălinescu. Cryptographically verified implementations for TLS. In *CCS'08*, pages 459–468. ACM, 2008.

[13] Karthikeyan Bhargavan, Cédric Fournet, and Andrew Gordon. Modular verification of security protocol code by typing. In *POPL'10*, pages 445–456. ACM, 2010.

[14] Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM TOPLAS*, 31(1), 2008.

[15] Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *CSF'07*, pages 97–111. IEEE, 2007.

[16] Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, 2008.

[17] Bruno Blanchet. Using Horn clauses for analyzing security protocols. In *Formal Models and Techniques for Analyzing Security Protocols*, volume 5 of *Cryptology and Information Security Series*, pages 86–111. IOS Press, Amsterdam, 2011.

[18] Bruno Blanchet. Automatically verified mechanized proof of one-encryption key exchange. In *CSF'12*, pages 325–339, Los Alamitos, 2012. IEEE.

[19] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations. In *ARES'12*, pages 65–74, Los Alamitos, 2012. IEEE.

[20] David Cadé and Bruno Blanchet. From computationally-proved protocol specifications to implementations and application to SSH. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):4–31, March 2013.

[21] David Cadé and Bruno Blanchet. Proved generation of implementations from computationally-secure protocol specifications. In David Basin and John Mitchell, editors, *2nd Conference on Principles of Security and Trust (POST 2013)*, volume 7796 of *LNCS*, pages 63–82, Rome, Italy, March 2013. Springer.

[22] Sagar Chaki and Anupam Datta. ASPIER: An automated framework for verifying security protocol implementations. In *CSF'09*, pages 172–185, Los Alamitos, 2009. IEEE.

[23] Hubert Comon-Lundh and Véronique Cortier. Computational soundness of observational equivalence. In *CCS'08*, pages 109–118. ACM, 2008.

[24] Véronique Cortier, Heinrich Hördegen, and Bogdan Warinschi. Explicit randomness is not necessary when modeling probabilistic encryption. In *ICS'06*, volume 186 of *ENTCS*, pages 49–65, Amsterdam, 2006. Elsevier.

[25] Véronique Cortier and Bogdan Warinschi. Computationally sound, automated proofs for security protocols. In *ESOP'05*, volume 3444 of *LNCS*, pages 157–171. Springer, 2005.

[26] Judicaël Courant, Marion Daubignard, Cristian Ene, Pascal Lafourcade, and Yassine Lakhnech. Automated proofs for asymmetric encryption. In *Concurrency, Compositionality, and Correctness*, volume 5930 of *LNCS*, pages 300–321. Springer, 2010.

[27] Cas J. F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* PhD thesis, Eindhoven University of Technology, 2006.

[28] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, 1983.

[29] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *CSF'11*, pages 3–17, Los Alamitos, 2011. IEEE.

[30] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *CCS'11*, pages 341–350, New York, 2011. ACM.

[31] `http://msr-inria.inria.fr/projects/sec/fs2cv/`.

[32] Jean Goubault-Larrecq and Fabrice Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI'05*, volume 3385 of *LNCS*, pages 363–379. Springer, 2005.

[33] Jan Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *ASE'06*, pages 167–176. IEEE, 2006.

[34] Tero Kivinen and Mika Kojo. RFC 3526: More modular exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). `http://www.ietf.org/rfc/rfc3526.txt`, 2003.

[35] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS'96*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[36] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[37] Giuseppe Milicia. χ-spaces: Programming security protocols. In *NWPT'02*, 2002.

[38] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[39] `http://caml.inria.fr/`.

[40] Nicholas O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. In *FCS-ARSPA-WITS'08*, 2008.

[41] Scott Owens. A sound semantics for OCaml light. In Sophia Drossopoulou, editor, *ESOP'08*, volume 4960 of *LNCS*, pages 1–15, Heidelberg, 2008. Springer.

[42] Kenneth G. Paterson and Gaven J. Watson. Plaintext-dependent decryption: A formal security treatment of SSH-CTR. In *Eurocrypt 2010*, volume 6110 of *LNCS*, pages 345–361. Springer, 2010. Full version available at `http://eprint.iacr.org/2010/095`.

[43] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *ISCC'07*, pages 839–844. IEEE, 2007.

[44] Alfredo Pironti and Riccardo Sisto. Provably correct Java implementations of spi calculus security protocols specifications. *Computers and Security*, 29(3):302–314, 2010.

[45] Erik Poll and Aleksy Schubert. Verifying an implementation of SSH. In *WITS'07*, 2007.

[46] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2Java: Automatic cryptographic protocol Java code generation from spi calculus. In *AINA'04*, volume 1, pages 400–405. IEEE, 2004.

[47] Dawn Song, Adrian Perrig, and Doantam Phan. AGVI—Automatic Generation, Verification, and Implementation of security protocols. In *CAV'01*, volume 2102 of *LNCS*, pages 241–245. Springer, 2001.

[48] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. In *ICFP'11*, pages 266–278, New York, 2011. ACM.

[49] Bogdan Warinschi. A computational analysis of the Needham-Schroeder-(Lowe) protocol. In *16th Computer Security Foundations Workshop (CSFW'03)*, pages 248–262. IEEE, 2003.

[50] Tatu Ylönen. RFC 4251: The Secure Shell (SSH) Protocol Architecture. `http://www.ietf.org/rfc/rfc4251.txt`, 2006.

[51] Tatu Ylönen. RFC 4252: The Secure Shell (SSH) Authentication Protocol. `http://www.ietf.org/rfc/rfc4252.txt`, 2006.

[52] Tatu Ylönen. RFC 4253: The Secure Shell (SSH) Transport Layer Protocol. http://www.ietf.org/rfc/rfc4253.txt, 2006.

[53] Tatu Ylönen. RFC 4254: The Secure Shell (SSH) Connection Protocol. http://www.ietf.org/rfc/rfc4254.txt, 2006.

Appendix A

Full Proof of Translation Correctness

In this chapter, we prove Lemma 8.15, that shows the correctness of the translation.

Let us first prove some auxiliary lemmas.

Lemma A.1 (Write file) *Let $\mathcal{C}$ be an OCaml configuration. If $\mathcal{C}_{Th}(\mathcal{C}) = \langle env, \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]), stack, store\rangle$, $env(\mathbb{G}_{\mathsf{var}}(x)) = \mathbb{G}_{\mathsf{val}T_x}(a)$, $env \supseteq env_{\mathsf{prim}}$, and $\mathcal{C}_{globalstore}(\mathcal{C}) \supseteq globalstore(E, \mathcal{T})$, then we have $\mathcal{C} \longrightarrow^* \mathcal{C}'$ where*

- $\mathcal{C}' = \mathcal{C}[\mathsf{Th} \mapsto \langle env, (), stack, store'\rangle, \mathsf{globalstore} \mapsto globalstore']$,
- $store' \supseteq store$,
- $globalstore' \supseteq globalstore(E[x[\widetilde{a}] \mapsto a], \mathcal{T})$,
- $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$ *for all* $l \notin S_{priv}$.

Proof If $(x[\widetilde{a}], f) \in \mathsf{files}$ for some f, then we have $x[\widetilde{a}] = x[\,]$ and $\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]) = (f := \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x))$, so

$$\begin{aligned}
\mathcal{C} &\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{ser}}(T_x)\ \mathbb{G}_{\mathsf{var}}(x)), stack', store\rangle] \\
&\quad \text{where } stack' \stackrel{\text{def}}{=} (env, f := [\cdot]) :: stack \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{ser}}(T_x))\ \mathbb{G}_{\mathsf{val}T_x}(a), stack', store\rangle] \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env', \mathsf{ser}(T_x, a), stack', store'\rangle] \qquad \text{by Proposition 8.5} \\
&\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, f := \mathsf{ser}(T_x, a), stack, store'\rangle] \\
&\to \mathcal{C}' = \mathcal{C}[\mathsf{Th} \mapsto \langle env, (), stack, store'\rangle, \mathsf{globalstore} \mapsto globalstore']
\end{aligned}$$

where

$$\begin{aligned}
globalstore' &\stackrel{\text{def}}{=} \mathcal{C}_{globalstore}(\mathcal{C})[f \mapsto \mathsf{ser}(T_x, a)] \\
&\supseteq globalstore(E, \mathcal{T})[f \mapsto \mathsf{ser}(T_x, a)] \\
&\supseteq globalstore(E[x[\,] \mapsto a], \mathcal{T})\,.
\end{aligned}$$

The modified location f is in S_{priv}, so for all $l \notin S_{priv}$, we have $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$. We have $store' \supseteq store$ by Proposition 8.5.

Otherwise, we have $\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]) = ()$ and $\mathcal{C}' = \mathcal{C}$, so

$$\begin{aligned} \mathit{globalstore}' &= \mathcal{C}_{\mathit{globalstore}}(\mathcal{C}) \\ &\supseteq \mathit{globalstore}(E, \mathcal{T}) = \mathit{globalstore}(E[x[\widetilde{a}] \mapsto a], \mathcal{T}) \,, \end{aligned}$$

and for all $l \notin S_{priv}$, we have $\mathit{globalstore}'(l) = \mathcal{C}_{\mathit{globalstore}}(\mathcal{C})(l)$. □

Definition A.2 (Deserialized OCaml values for tables) *Let Tbl be a table of type $T_1 \times \cdots \times T_l$. The OCaml value that corresponds to an element $(b_1, \ldots, b_l)$ of this table is*

$$\mathbb{G}_{\mathsf{val}T_1,\ldots,T_l}(b_1, \ldots, b_l) \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_l}(a_l)) \,.$$

Let $t = [a_1; \ldots; a_k]$ be a list of elements of table Tbl. The corresponding OCaml list is

$$\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t) \stackrel{\text{def}}{=} [\mathbb{G}_{\mathsf{val}T_1,\ldots,T_l}(a_1); \ldots; \mathbb{G}_{\mathsf{val}T_1,\ldots,T_l}(a_k)] \,.$$

Definition A.3 (Function filter) *Let E be a CryptoVerif environment, M a CryptoVerif boolean term, $(x_1, \ldots, x_k)$ a tuple of variables, and t a list of tuples of CryptoVerif values of type $T_{x_1} \times \cdots \times T_{x_k}$. We let $\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t)$ be the list of tuples $(a_1, \ldots, a_k)$ in t such that the term M is true when the variables $x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]$ are bound to $a_1, \ldots, a_k$ in the environment E, respectively:*

$$\begin{aligned} &\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t) \stackrel{\text{def}}{=} \\ &\quad [(a_1, \ldots, a_k) \in t \mid E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k], M \Downarrow \text{true}] \end{aligned}$$

Let us recall that our fold function on lists $\mathbb{G}_{\mathsf{fold}}$ is defined in Figure 3.1 as follows:

$$\mathbb{G}_{\mathsf{fold}} \stackrel{\text{def}}{=} f \to \mathsf{function}\ a \to \mathsf{function}\ [\,] \to a \mid x :: l \to f\ (\mathit{fold}\ f\ a\ l)\ x$$

Lemma A.4 *Suppose that*

$$\begin{aligned} \mathit{Th} &= \langle \mathit{env}, \mathit{fold}\ c\ [\,]\ (\mathbb{G}_{\mathsf{tbl}}(\mathit{Tbl}, t)), \mathit{stack}, \mathit{store} \rangle \\ c &= \mathsf{function}[\mathit{env}', \\ &\qquad a \to \mathsf{function}\ x \to \mathsf{try}\ (c'\ x) :: a\ \mathsf{with}\ \mathsf{Match\_failure} \to a] \\ c' &= \mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M) \\ \mathit{env}' &\supseteq \mathit{env}_{\mathsf{prim}} \cup \{\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T_x}(b) \mid x[\widetilde{a}'] \in \mathsf{fv}(M) \setminus \{x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]\}, \\ &\qquad\qquad E(x[\widetilde{a}']) = b\} \\ \mathit{env}(\mathit{fold}) &= \mathit{env}'(\mathit{fold}) = \mathsf{letrec}[\mathit{env}_0, \{\mathit{fold} \mapsto \mathbb{G}_{\mathsf{fold}}\}\ \mathsf{in}\ \mathit{fold}] \\ &\qquad \textit{where } t \textit{ is a list of CryptoVerif values of type } T_{x_1} \times \cdots \times T_{x_k} \\ &\qquad \textit{and all occurrences of } x_1, \ldots, x_k \textit{ in } M \textit{ have indices } \widetilde{a}. \end{aligned}$$

Then $\mathit{Th} \to^* \mathit{Th}'$ such that

$$\mathit{Th}' = \langle \mathit{env}'', \mathbb{G}_{\mathsf{tbldeser}}(\mathit{Tbl}, \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t)), \mathit{stack}, \mathit{store}' \rangle$$

for some env'' and store' such that $\mathit{store}' \supseteq \mathit{store}$.

Proof We prove this lemma by induction on the length of t.

In the base case, $t = [\,]$, so $\mathbb{G}_{\mathsf{tbl}}(Tbl, t) = [\,]$, and

$$
\begin{aligned}
Th &= \langle env, fold\ c\ [\,]\ [\,], stack, store\rangle \\
&\to^* \langle env_1, (\mathsf{match}\ c\ \mathsf{with}\ \mathbb{G}_{\mathsf{fold}})\ [\,]\ [\,], stack, store\rangle \\
&\quad \text{where } env_1 \stackrel{\text{def}}{=} env_0[fold \mapsto env'(fold)] \\
&\to^* \langle env'', \mathsf{match}\ [\,]\ \mathsf{with}\ [\,] \mapsto a \mid x :: l \to f\ (fold\ f\ a\ l)\ x, stack, store\rangle \\
&\quad \text{where } env'' \stackrel{\text{def}}{=} env_1[f \mapsto c, a \mapsto [\,]] \\
\to^*\ Th' &\stackrel{\text{def}}{=} \langle env'', [\,], stack, store\rangle
\end{aligned}
$$

For any $E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}])$, we have $\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), [\,]) = [\,]$. So the lemma is correct for the base case.

In the inductive case, let $t = b :: t'$. Let $y = \mathbb{G}_{\mathsf{tblel}}(Tbl, b)$ and $l' = \mathbb{G}_{\mathsf{tbl}}(Tbl, t')$, so $\mathbb{G}_{\mathsf{tbl}}(Tbl, t) = y :: l'$. Let $b = (a_1, \ldots, a_k)$ and $y = (d_1, \ldots, d_k)$, where each d_i corresponds to a_i. Let $(d'_1, \ldots, d'_k) = \mathbb{G}_{\mathsf{val}T_1,\ldots,T_k}(b)$, where Tbl is a table of type $T_1 \times \cdots \times T_k$.

$$
\begin{aligned}
Th &= \langle env, fold\ c\ [\,]\ (y :: l'), stack, store\rangle \\
&\to^* \langle env_1, (\mathsf{match}\ c\ \mathsf{with}\ \mathbb{G}_{\mathsf{fold}})\ [\,]\ (y :: l'), stack, store\rangle \\
&\quad \text{where } env_1 \stackrel{\text{def}}{=} env_0[fold \mapsto env'(fold)] \\
&\to^* \langle env_2, \mathsf{match}\ y :: l'\ \mathsf{with}\ [\,] \mapsto a \mid x :: l \to f\ (fold\ f\ a\ l)\ x, stack, store\rangle \\
&\quad \text{where } env_2 \stackrel{\text{def}}{=} env_1[f \mapsto c, a \mapsto [\,]] \\
&\to^* \langle env_3, f\ (fold\ f\ a\ l)\ x, stack, store\rangle \\
&\quad \text{where } env_3 \stackrel{\text{def}}{=} env_2[x \mapsto y, l \mapsto l'] \\
&\to^* \langle env_3, fold\ f\ a\ l, stack_1, store\rangle \text{ where } stack_1 \stackrel{\text{def}}{=} (env_3, f\ [\cdot]\ y) :: stack \\
&\qquad\qquad \text{(arguments are evaluated from right to left)} \\
&\to^* \langle env_3, fold\ c\ [\,]\ l', stack_1, store\rangle \\
&\to^* \langle env_4, \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''), stack_1, store_1\rangle \qquad \text{by induction hypothesis} \\
&\quad \text{where } t'' \stackrel{\text{def}}{=} \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t') \\
&\to \langle env_3, f\ (\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''))\ y, stack, store_1\rangle \\
&\to^* \langle env_3, c\ (\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''))\ y, stack, store_1\rangle \\
&\to^* \langle env_5, \mathsf{try}\ (c'\ x) :: a\ \mathsf{with}\ \mathsf{Match\_failure} \to a, stack, store_1\rangle \\
&\quad \text{where } env_5 \stackrel{\text{def}}{=} env'[a \mapsto \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''), x \mapsto y] \\
&\to \langle env_5, (c'\ x) :: a, stack_2, store_1\rangle \\
&\quad \text{where } stack_2 \stackrel{\text{def}}{=} (env_5, \mathsf{try}\ [\cdot]\ \mathsf{with}\ \mathsf{Match\_failure} \to a) :: stack \\
&\to \langle env_5, c'\ x, stack_3, store_1\rangle \text{ where } stack_3 \stackrel{\text{def}}{=} (env_5, [\cdot] :: a) :: stack_2 \\
&\to^* \langle env_5[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto d_1, \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto d_k], \\
&\quad \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x_1) = \mathbb{G}_{\mathsf{deser}}(T_{x_1})\ \mathbb{G}_{\mathsf{var}}(x_1)\ \mathsf{in} \ldots \\
&\quad \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x_k) = \mathbb{G}_{\mathsf{deser}}(T_{x_k})\ \mathbb{G}_{\mathsf{var}}(x_k)\ \mathsf{in} \\
&\quad \mathsf{if}\ (\mathbb{G}_{\mathsf{M}}(M))\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{else\ raise\ Match\_failure}, \\
&\quad stack_3, store_1\rangle
\end{aligned}
$$

$$\begin{aligned}
&\to^* \langle env_6, \\
&\quad \mathsf{if}\ (\mathbb{G}_{\mathsf{M}}(M))\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{else\ raise\ Match_failure}, \\
&\quad stack_3, store_2\rangle \text{ where } env_6 \stackrel{\text{def}}{=} env_5[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto d'_1, \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto d'_k] \\
&\qquad\qquad \text{by Proposition 8.5 applied } k \text{ times} \\
\to\ Th_1 &\stackrel{\text{def}}{=} \langle env_6, \mathbb{G}_{\mathsf{M}}(M), stack_4, store_2\rangle \\
&\quad \text{where } stack_4 \stackrel{\text{def}}{=} (env_6, \mathsf{if}\ [\cdot]\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) \\
&\qquad\qquad \mathsf{else\ raise\ Match_failure}) :: stack_3
\end{aligned}$$

The environment env_6 contains env_{prim} and $env(E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k], M)$. Let r be the CryptoVerif value such that $E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k], M \Downarrow r$. So by Lemma 8.13,

$$Th_1 \to^* Th_2 \stackrel{\text{def}}{=} \langle env_7, \mathbb{G}_{\mathsf{val}bool}(r), stack_4, store_3\rangle$$

and by Lemma 8.13, Proposition 8.5, and the induction hypothesis, we have $store_3 \supseteq store_2 \supseteq store_1 \supseteq store$.

Let us suppose that $r = \text{true}$. In this case, $\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t) = b :: \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t')$. Let us denote $t''' \stackrel{\text{def}}{=} \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t)$.

$$\begin{aligned}
Th_2 &= \langle env_7, \text{true}, stack_4, store_3\rangle \\
&\to^* \langle env_6, \mathsf{if}\ \text{true}\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{else\ raise\ Match_failure}, \\
&\quad stack_3, store_3\rangle \\
&\to \langle env_6, (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)), stack_3, store_3\rangle \\
&\to^* \langle env_6, (d'_1, \ldots, d'_k), stack_3, store_3\rangle \\
&\to \langle env_5, (d'_1, \ldots, d'_k) :: a, stack_2, store_3\rangle \\
&\to^* \langle env_5, \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t'''), stack_2, store_3\rangle \\
&\quad \text{since } \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''') = (d'_1, \ldots, d'_k) :: (\mathbb{G}_{\mathsf{tbldeser}}(Tbl, t'')) \\
&\to^* \langle env_5, \mathsf{try}\ \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''')\ \mathsf{with\ Match_failure} \to a, stack, store_3\rangle \\
\to\ Th' &\stackrel{\text{def}}{=} \langle env_5, \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t'''), stack, store_3\rangle
\end{aligned}$$

So the lemma is correct in this case.

Let us now suppose that $r = \text{false}$. In this case, $\mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t) = \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), t')$.

$$\begin{aligned}
Th_2 &= \langle env_7, \text{false}, stack_4, store_3\rangle \\
&\to^* \langle env_6, \mathsf{if}\ \text{false}\ \mathsf{then}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k))\ \mathsf{else\ raise\ Match_failure}, \\
&\quad stack_3, store_3\rangle \\
&\to \langle env_6, \mathsf{raise\ Match_failure}, stack_3, store_3\rangle \\
&\to^* \langle env_5, \mathsf{try\ raise\ Match_failure\ with\ Match_failure} \to a, stack, store_3\rangle \\
&\to^* \langle env_5, a, stack, store_3\rangle \\
\to\ Th' &\stackrel{\text{def}}{=} \langle env_5, \mathbb{G}_{\mathsf{tbldeser}}(Tbl, t''), stack, store_3\rangle
\end{aligned}$$

As in the previous case, the lemma is also correct in this case. □

Proof (of Lemma 8.15) Let us prove this lemma by looking at each case.

- The random number generation construct:

  On the CryptoVerif side, for each element b of type T, we have the following reduction:

  $$E, x[\widetilde{a}] \stackrel{R}{\leftarrow} T; P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_{1/|T|} E[x[\widetilde{a}] \mapsto b], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$$

  The variable $x[\widetilde{a}]$, bound in P, is free in P'.

  On the OCaml side, we have, for all $b \in T$,

  $$\begin{aligned}
  \mathcal{C} &= \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{random}}(T)\ ()\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P')), \\
  &\qquad\qquad stack, store\rangle] \\
  &\rightarrow \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{random}}(T)\ (), stack', store\rangle] \\
  &\qquad \text{where } stack' \stackrel{\text{def}}{=} (env, \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = [\cdot]\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'))) \\
  &\qquad\qquad :: stack \\
  &\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{random}}(T))\ (), stack', store\rangle] \\
  &\rightarrow^*_{1/|T|} \mathcal{C}[\mathsf{Th} \mapsto \langle env', \mathbb{G}_{\mathsf{val}T}(b), stack', store'\rangle] \qquad \text{by Proposition 8.5} \\
  &\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'), stack, store'\rangle] \\
  &\qquad \text{where } env'' \stackrel{\text{def}}{=} env[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)] \\
  &\rightarrow \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]), ([\cdot]; \mathbb{G}(P')) :: stack, store'\rangle] \\
  &\rightarrow^* \mathcal{C}[\mathsf{Th} \mapsto \langle env'', (); \mathbb{G}(P'), stack, store''\rangle, \mathsf{globalstore} \mapsto globalstore'] \\
  &\qquad\qquad \text{by Lemma A.1} \\
  \rightarrow \mathcal{C}' &= \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}(P'), stack, store''\rangle, \mathsf{globalstore} \mapsto globalstore']
  \end{aligned}$$

  This sequence of reductions describes the set of traces $\mathcal{CTS}_b$ that corresponds to the CryptoVerif reduction that adds to its environment the value b bound to $x[\widetilde{a}]$. We have $\Pr[\mathcal{CTS}_b] = 1/|T|$.

  Let $E' \stackrel{\text{def}}{=} E[x[\widetilde{a}] \mapsto b]$. We have $env(E', P') = env(E, P)[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(a)]$, so $env'' \supseteq env_{\mathsf{prim}} \cup env(E', P')$. By Lemma A.1, we have $globalstore' \supseteq globalstore(E', \mathcal{T})$ and $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$ for all $l \notin S_{priv}$. By Lemma A.1 and Proposition 8.5, we have $store'' \supseteq store' \supseteq store$. Events are unchanged on both sides. So this construct satisfies the lemma.

- The assignment construct:

  On the CryptoVerif side, let us suppose that $E, M \Downarrow b$. We have the following reduction:

  $$E, x[\widetilde{a}] \leftarrow M; P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow E[x[\widetilde{a}] \mapsto b], P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}$$

  Let us denote $T = T_M$. The variable $x[\widetilde{a}]$, bound in P, is free in P'.

  On the OCaml side, we have:

  $$\begin{aligned}
  &\mathcal{C} = \mathcal{C}[\mathsf{Th} \mapsto Th] \text{ where } Th \stackrel{\text{def}}{=} \\
  &\qquad \langle env, \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P')), stack, store\rangle
  \end{aligned}$$

$$\begin{aligned}
&\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{M}}(M), stack', store\rangle] \\
&\quad \text{where } stack' \stackrel{\text{def}}{=} (env, \mathsf{let}\ \mathbb{G}_{\mathsf{var}}(x) = [\cdot]\ \mathsf{in}\ (\mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'))) \\
&\qquad\qquad :: stack \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{val}T}(b), stack', store'\rangle] \qquad \text{by Lemma 8.13} \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env', \mathbb{G}_{\mathsf{file}}(x[\widetilde{a}]); \mathbb{G}(P'), stack, store'\rangle] \\
&\quad \text{where } env' \stackrel{\text{def}}{=} env[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)] \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env', (); \mathbb{G}(P'), stack, store''\rangle, \mathsf{globalstore} \mapsto globalstore'] \\
&\qquad\qquad \text{by Lemma A.1} \\
&\to \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env', \mathbb{G}(P'), stack, store''\rangle, \mathsf{globalstore} \mapsto globalstore']
\end{aligned}$$

This sequence of reductions describes the set of traces $\mathcal{CTS}_1$ that corresponds to the CryptoVerif reduction. We have $\Pr[\mathcal{CTS}_1] = 1$.

Let $E' \stackrel{\text{def}}{=} E[x[\widetilde{a}] \mapsto b]$. We have $env(E', P') = env(E, P)[\mathbb{G}_{\mathsf{var}}(x) \mapsto \mathbb{G}_{\mathsf{val}T}(b)]$, so $env' \supseteq env_{\mathsf{prim}} \cup env(E', P')$. By Lemma A.1, we have $globalstore' \supseteq globalstore(E', \mathcal{T})$ and $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$ for all $l \notin S_{priv}$. By Lemmas A.1 and 8.13, we have $store'' \supseteq store' \supseteq store$. Events are unchanged on both sides. So this construct satisfies the lemma.

- The conditional construct:

  On the CryptoVerif side, let us suppose that $E, M \Downarrow \text{true}$. The same reasoning can be applied in the case that $E, M \Downarrow \text{false}$. We have the following reduction:

  $$E, \mathsf{if}\ M\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to E, P_1, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}.$$

  On the OCaml side, we implement the CryptoVerif *bool* type with booleans in OCaml, and we have $\mathbb{G}_{\mathsf{val}bool}(\text{true}) = \text{true}$ and $\mathbb{G}_{\mathsf{val}bool}(\text{false}) = \text{false}$. Let $Th = \mathcal{C}_{Th}(\mathcal{C})$.

  $$\begin{aligned}
  Th &= \langle env, \mathsf{if}\ \mathbb{G}_{\mathsf{M}}(M)\ \mathsf{then}\ \mathbb{G}(P_1)\ \mathsf{else}\ \mathbb{G}(P_2), stack, store\rangle \\
  &\to \langle env, \mathbb{G}_{\mathsf{M}}(M), stack', store\rangle \\
  &\quad \text{where } stack' \stackrel{\text{def}}{=} (env, \mathsf{if}\ [\cdot]\ \mathsf{then}\ \mathbb{G}(P_1)\ \mathsf{else}\ \mathbb{G}(P_2)) :: stack \\
  &\to^* \langle env, \text{true}, stack', store'\rangle \qquad \text{by Lemma 8.13} \\
  &\to \langle env, \mathsf{if}\ \text{true}\ \mathsf{then}\ \mathbb{G}(P_1)\ \mathsf{else}\ \mathbb{G}(P_2), stack, store'\rangle \\
  &\to Th' \stackrel{\text{def}}{=} \langle env, \mathbb{G}(P_1), stack, store'\rangle
  \end{aligned}$$

  By (Globalstore1) and (Toplevel), we obtain $\mathcal{C} \to^* \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto Th']$. This sequence of reductions describes the set of traces $\mathcal{CTS}_1$ that corresponds to the CryptoVerif reduction. We have $\Pr[\mathcal{CTS}_1] = 1$.

  The CryptoVerif environment E and tables $\mathcal{T}$, the OCaml environment *env* and global store *globalstore*, and the events on both sides are unchanged. By Lemma 8.13, we have $store' \supseteq store$, so this construct satisfies the lemma.

- The insert construct:

  On the CryptoVerif side, let us suppose that $E, M_i \Downarrow a_i$. We have the following reduction:

  $$E, \mathsf{insert}\ Tbl(M_1, \ldots, M_k); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to E, P', \mathcal{T}', \mathcal{Q}, \mathcal{R}, \mathcal{E}\,,$$

  where $\mathcal{T}' \stackrel{\text{def}}{=} \mathcal{T}[Tbl \mapsto (a_1, \ldots, a_k) :: \mathcal{T}(Tbl)]$. Let the type of the table Tbl be $T_1 \times \cdots \times T_k$.

  On the OCaml side, there exists a unique f such that $(Tbl, f) \in \mathsf{tables}$. Let $globalstore = \mathcal{C}_{globalstore}(\mathcal{C})$. Since $globalstore \supseteq globalstore(E, \mathcal{T})$, we have

  $$globalstore(f) = t \text{ where } t \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{tbl}}(Tbl, \mathcal{T}(Tbl))\,.$$

  Let $t' \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{tblel}}(Tbl, (a_1, \ldots, a_k)) :: t$. By definition of $\mathbb{G}_{\mathsf{tbl}}$, we have $t' = \mathbb{G}_{\mathsf{tbl}}(Tbl, \mathcal{T}'(Tbl))$. Let $globalstore' = globalstore[f \mapsto t']$.

  $$\begin{aligned}
  \mathcal{C} &= \mathcal{C}[\mathsf{Th} \mapsto \langle env, f := e :: (!f); \mathbb{G}(P'), stack, store\rangle] \\
  &\quad \text{where } e \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{ser}}(T_1)\ \mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{ser}}(T_k)\ \mathbb{G}_{\mathsf{M}}(M_k)) \\
  &\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, e :: t, stack', store\rangle] \\
  &\quad \text{where } stack' \stackrel{\text{def}}{=} (env, f := [\cdot]) :: (env, [\cdot]; \mathbb{G}(P')) :: stack \\
  &\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}_{\mathsf{tblel}}(Tbl, (a_1, \ldots, a_k)) :: t, stack', store'\rangle] \\
  &\qquad\qquad \text{by Lemma 8.13 and Proposition 8.5} \\
  &\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, f := t', (env, [\cdot]; \mathbb{G}(P')) :: stack, store'\rangle] \\
  &\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, (); \mathbb{G}(P'), stack, store'\rangle, \mathsf{globalstore} \mapsto globalstore'] \\
  &\to \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}(P'), stack, store'\rangle, \mathsf{globalstore} \mapsto globalstore']
  \end{aligned}$$

  This sequence of reductions describes the set of traces $\mathcal{CTS}_1$ that corresponds to the CryptoVerif reduction. We have $\Pr[\mathcal{CTS}_1] = 1$.

  The global store is modified so that $globalstore' \supseteq globalstore(E, \mathcal{T}')$ and $globalstore'(l) = \mathcal{C}_{globalstore}(\mathcal{C})(l)$ for all $l \notin S_{priv}$, and the environments and events are unchanged on both sides. Moreover, by Proposition 8.5 and Lemma 8.13, we have $store' \supseteq store$, so this construct satisfies the lemma.

- The get construct:

  On the CryptoVerif side, let us consider a CryptoVerif configuration such that its program is

  $$P = \mathsf{get}\ Tbl(x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}])\ \mathsf{suchthat}\ M\ \mathsf{in}\ P'\ \mathsf{else}\ P''\,.$$

  Let the type of the table Tbl be $T_1 \times \cdots \times T_k$.

  We have two cases depending on whether there is a value in the table Tbl that satisfies M or not. Let $l' \stackrel{\text{def}}{=} \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), \mathcal{T}(Tbl)) = [b_1, \ldots, b_m]$. This list contains every element of $\mathcal{T}(Tbl)$ such that M is true.

If l' is empty, then:

$$E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to E, P'', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}.$$

If l' is not empty, then there is a reduction for each element $b = (a_1, \ldots, a_k)$ in l',

$$E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to_{p_b} E_b, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}\,,$$

where $p_b \stackrel{\text{def}}{=} \sum_{\{j \in \{1,\ldots,m\} | b_j = b\}} \text{among}(\{1, \ldots, m\}, j)$ and $E_b \stackrel{\text{def}}{=} E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k]$.

On the OCaml side, let us denote

$$\begin{aligned}
e &\stackrel{\text{def}}{=} \mathsf{if}\ l = [\,]\ \mathsf{then}\ \mathbb{G}(P')\ \mathsf{else}\\
&\quad \mathsf{let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = \mathsf{random}_l\ l\ \mathsf{in}\\
&\quad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P))\\
e_1 &\stackrel{\text{def}}{=} \mathsf{try}\ (\mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M)\ x) :: a\ \mathsf{with}\ \mathsf{Match_failure} \to a
\end{aligned}$$

There exists a unique f such that $(Tbl, f) \in \mathsf{tables}$, and we have

$$\begin{aligned}
\mathcal{C} &= \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathsf{let}\ l = e_2\ \mathsf{in}\ e, stack, store\rangle]\\
&\quad \text{where } e_2 \stackrel{\text{def}}{=} \text{read_table}(f, \mathbb{G}_{\mathsf{test}}((x_1, \ldots, x_k), M))\\
&\qquad = \mathsf{let\ rec}\ fold = \mathsf{function}\ \mathbb{G}_{\mathsf{fold}}\ \mathsf{in}\\
&\qquad\quad fold\ (\mathsf{function}\ a \to\ x \to\ e_1)\ [\,]\ !f\\
&\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, e_2, stack', store\rangle]\\
&\quad \text{where } stack'' \stackrel{\text{def}}{=} (env, \mathsf{let}\ l = [\cdot]\ \mathsf{in}\ e) :: stack\\
&\to \mathcal{C}[\mathsf{Th} \mapsto \langle env', fold\ (\mathsf{function}\ a \to\ x \to\ e_1)\ [\,]\ !f, stack', store\rangle]\\
&\quad \text{where } env' \stackrel{\text{def}}{=} env[fold \mapsto \mathsf{letrec}[env, \{fold \mapsto \mathbb{G}_{\mathsf{fold}}\}\ \mathsf{in}\ fold]]\\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env', e_3, stack', store\rangle]\\
&\quad \text{where } e_3 \stackrel{\text{def}}{=}\\
&\qquad fold\ \mathsf{function}[env', a \to\ \mathsf{function}\ x \to\ e_1]\ [\,]\ \mathbb{G}_{\mathsf{tbl}}(Tbl, \mathcal{T}(Tbl))\\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}_{\mathsf{tbldeser}}(Tbl, l'), stack', store'\rangle] \qquad \text{by Lemma A.4}\\
&\quad \text{since } l' = \mathsf{filter}(E, M, (x_1[\widetilde{a}], \ldots, x_k[\widetilde{a}]), \mathcal{T}(Tbl))\\
&\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathsf{let}\ l = \mathbb{G}_{\mathsf{tbldeser}}(Tbl, l')\ \mathsf{in}\ e, stack, store'\rangle]\\
\to \mathcal{C}_1 &\stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env'', e, stack, store'\rangle]\\
&\quad \text{where } env'' \stackrel{\text{def}}{=} env[l \mapsto \mathbb{G}_{\mathsf{tbldeser}}(Tbl, l')]
\end{aligned}$$

Now, if l' is empty, then $env''(l) = [\,]$, so

$$\mathcal{C}_1 \to^* \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env'', \mathbb{G}(P''), stack, store'\rangle]$$

The sequence of reductions $\mathcal{C} \to^* \mathcal{C}_1 \to^* \mathcal{C}'$ describes the set of traces $\mathcal{CTS}_1$ that corresponds to the unique CryptoVerif reduction that can happen when l' is empty. We have $\Pr[\mathcal{CTS}_1] = 1$.

The CryptoVerif environment E is unchanged and the OCaml environment env'' is an extension of env, so we have $env'' \supseteq env_{\mathsf{prim}} \cup env(E, P'')$. The CryptoVerif tables, the global store, and the events on both sides are unchanged. By Lemma A.4, we have $store' \supseteq store$. So, in this case, the get construct satisfies the lemma.

If l' is not empty, then let $b = (a_1, \ldots, a_k)$ be any element of l', and let $v = \mathbb{G}_{\mathsf{val}T_1,\ldots,T_k}(Tbl, b) = (\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_k}(a_k))$. We have $env''(l) = \mathbb{G}_{\mathsf{tbldeser}}(Tbl, l')$. Let $env''(l) = [v_1; \ldots; v_m]$. The set $S \stackrel{\text{def}}{=} \{j \in \{1, \ldots, m\} \mid v = v_j\}$ is equal to the set $\{j \in \{1, \ldots, m\} \mid b = b_j\}$, because the function $b \mapsto \mathbb{G}_{\mathsf{val}T_1,\ldots,T_k}(Tbl, b)$ is injective. Hence, we have $p_b = \sum_{j \in S} \mathrm{among}(\{1, \ldots, m\}, j)$.

$$
\begin{aligned}
\mathcal{C}_1 &\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env'', e_4, stack, store' \rangle] \\
&\quad \text{where } e_4 \stackrel{\text{def}}{=} \mathsf{let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = \mathsf{random}_l\ l\ \mathsf{in} \\
&\qquad\qquad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P')) \\
&\to^*_{p_b} \mathcal{C}[\mathsf{Th} \mapsto \langle env'', e_5, stack, store'' \rangle] \qquad \text{by Proposition 8.5} \\
&\quad \text{where } e_5 \stackrel{\text{def}}{=} \mathsf{let}\ (\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k)) = v\ \mathsf{in} \\
&\qquad\qquad (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P')) \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env''', \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{a}]); \mathbb{G}(P'), stack, store'' \rangle] \\
&\quad \text{where } env''' \stackrel{\text{def}}{=} \\
&\qquad env''[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_k}(a_k)] \\
&\to^* \mathcal{C}'_b \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env''', \mathbb{G}(P'), stack, store''' \rangle, \mathsf{globalstore} \mapsto globalstore'] \\
&\qquad\qquad \text{by Lemma A.1}
\end{aligned}
$$

The sequence of reductions $\mathcal{C} \to^* \mathcal{C}_1 \to^* \mathcal{C}'_b$ describes the set of traces $\mathcal{CTS}_b$ that corresponds to the CryptoVerif reduction in which the element $b = (a_1, \ldots, a_k)$ of l' is chosen. We have $\Pr[\mathcal{CTS}_b] = p_b$.

By Lemma A.1, $globalstore' \supseteq globalstore(E_b, \mathcal{T})$, and $globalstore'$ and $\mathcal{C}_{globalstore}(\mathcal{C})$ are equal on all locations not in S_{priv}, since $E_b = E[x_1[\widetilde{a}] \mapsto a_1, \ldots, x_k[\widetilde{a}] \mapsto a_k]$. Since the OCaml environment is $env''' = env[l \mapsto \ldots, \mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_k}(a_k)]$, we have $env''' \supseteq env_{\mathsf{prim}} \cup env(E_b, P')$. The events are unchanged on both sides. By Lemma A.1, Proposition 8.5, and Lemma A.4, we have $store''' \supseteq store'' \supseteq store' \supseteq store$. So, in this case, the get construct also satisfies the lemma.

- The event construct:

  On the CryptoVerif side, let us suppose that $E, M_j \Downarrow a_j$ for all $j \leq l$. We have the following reduction:

  $$E, \mathsf{event}\ ev(M_1, \ldots, M_l); P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \to E, P', \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}'\,,$$

  where $\mathcal{E}' \stackrel{\text{def}}{=} ev(a_1, \ldots, a_l) :: \mathcal{E}$. Let us denote $T_1 \times \cdots \times T_l$ the type of the event ev.

  On the OCaml side, let us denote

  $$events' \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{ev}}(\mathcal{E}') = ev(\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_l}(a_l)) :: events\,.$$

We have

$$\begin{aligned}
\mathcal{C} &= \mathcal{C}[\mathsf{Th} \mapsto \langle env, e; \mathbb{G}(P'), stack, store\rangle] \\
&\quad \text{where } e \stackrel{\text{def}}{=} \mathsf{event}\ ev(\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_l)) \\
&\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, e, stack', store, \rangle] \\
&\quad \text{where } stack' \stackrel{\text{def}}{=} (env, [\cdot]; \mathbb{G}(P')) :: stack \\
&\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env, e', stack', store'\rangle] \qquad \text{by Lemma 8.13} \\
&\quad \text{where } e' \stackrel{\text{def}}{=} \mathsf{event}\ ev(\mathbb{G}_{\mathsf{val}T_1}(a_1), \ldots, \mathbb{G}_{\mathsf{val}T_l}(a_l)) \\
&\to \mathcal{C}[\mathsf{Th} \mapsto \langle env, (), stack', store'\rangle, \mathsf{events} \mapsto events'] \\
&\to^* \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env, \mathbb{G}(P'), stack, store'\rangle, \mathsf{events} \mapsto events']
\end{aligned}$$

This sequence of reductions describes the set of traces $\mathcal{CTS}_1$ that corresponds to the CryptoVerif reduction. We have $\Pr[\mathcal{CTS}_1] = 1$.

The CryptoVerif environment E and tables $\mathcal{T}$ and the OCaml environment *env* and global store *globalstore* are unchanged. We have $events' = \mathbb{G}_{\mathsf{ev}}(\mathcal{E}')$. By Lemma 8.13, we have $store' \supseteq store$, so this construct satisfies the lemma. □

Appendix B

Full Proof of Correspondence

This chapter presents the proof of Lemmas 8.33 and 8.34, that show how the invariant defined in Definition 8.32 holds when reducing.

Proof (of Lemma 8.33) Let $\mathfrak{C}_0^{\mathsf{cs}} \stackrel{\text{def}}{=} \mathfrak{C}^0(Q_0, \mathit{program}_0)$.

$$
\begin{aligned}
\mathfrak{C}_0^{\mathsf{cs}} &= \emptyset, \mathsf{let}\ x[\,] : \mathit{bitstring} = O_{\text{start}}()\ \mathsf{in}\ \mathsf{return}(x[\,])\ \mathsf{else}\ \mathsf{end}, \mathcal{T}_0, \mathcal{Q}_0, \emptyset, [\,] \\
&\quad \text{where } \mathcal{Q}_0 \stackrel{\text{def}}{=} \{Q_{\text{start}}(Q_0, \mathit{program}_0)\} \cup \bigcup_{a \leq N_{\mathsf{rand}+\mathsf{calls}}} \{Q_{\mathit{loop}}\{a/i'\}\} \\
&\qquad\qquad \cup \,\text{reduce}(Q_0) \\
&\rightsquigarrow \emptyset, P_1, \mathcal{T}_0, \mathcal{Q}_1, \mathcal{R}_1, [\,] \\
&\quad \text{where } \mathcal{Q}_1 \stackrel{\text{def}}{=} \mathcal{Q}_0 \setminus \{Q_{\text{start}}(Q_0, \mathit{program}_0)\}, \\
&\qquad\quad P_1 \stackrel{\text{def}}{=} s_0[\,] \leftarrow s_0(Q_0, \mathit{program}_0); P_2, \\
&\qquad\quad P_2 \stackrel{\text{def}}{=} \mathsf{let}\ r[\,] : T_{\mathcal{CS}} = \mathsf{loop}\ O_{\text{loop}}[1](s_0)\ \mathsf{in}\ \mathsf{end}\ \mathsf{else}\ \mathsf{end}, \\
&\qquad\quad \mathcal{R}_1 \stackrel{\text{def}}{=} [x[\,], \mathsf{return}(x[\,]), \mathsf{end}] \\
&\rightsquigarrow E_1, P_2, \mathcal{T}_0, \mathcal{Q}_1, \mathcal{R}_1, [\,] \\
&\quad \text{where } E_1 \stackrel{\text{def}}{=} \{s_0[\,] \mapsto s_0(Q_0, \mathit{program}_0)\} \\
&\rightsquigarrow E_1, P_3, \mathcal{T}_0, \mathcal{Q}_1, \mathcal{R}_1, [\,] \\
&\quad \text{where } P_3 \stackrel{\text{def}}{=} \mathsf{let}\ (r'_{1,r}[\,] : T_{\mathcal{CS}}, b_{1,r}[\,] : \mathit{bool}) = O_{\text{loop}}[1](s_0) \\
&\qquad\qquad \mathsf{in}\ P_{\mathit{return\text{-}loop}}(1)\ \mathsf{else}\ \mathsf{end} \\
&\rightsquigarrow E_2, P_{\mathit{loop}}\{1/i'\}, \mathcal{T}_0, \mathcal{Q}_2, \mathcal{R}_{\mathit{loop}}(1), [\,] \\
&\quad \text{where } E_2 \stackrel{\text{def}}{=} E_1[s[1] \mapsto s_0(Q_0, \mathit{program}_0)], \\
&\qquad\quad \mathcal{Q}_2 \stackrel{\text{def}}{=} \mathcal{Q}_1 \setminus \{Q_{\mathit{loop}}\{1/i'\}\}, \\
\rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} &\stackrel{\text{def}}{=} E_2, P_{\mathit{loop}}\{1/i'\}, \mathcal{T}_0, \mathcal{Q}_2, \mathcal{R}_{\mathit{loop}}(1), [\,], N_{\mathsf{steps}}, \mathcal{CS}_0 \\
&\quad \text{where } \mathcal{CS}_0 \stackrel{\text{def}}{=} ([\langle \emptyset, \mathit{program}_0, [\,], \emptyset \rangle], \mathit{globalstore}_0, 1), \mathcal{RI}_0, \emptyset
\end{aligned}
$$

We have

$$\mathcal{C}_0(Q_0, \mathit{program}_0) = [\langle \emptyset, \mathit{program}_0, [\,], \emptyset \rangle], \mathit{globalstore}_0, 1, \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q_0), [\,].$$

Let $\mathcal{CT}$ be the trace that consists only of the configuration $\mathcal{C}_0(Q_0, \mathit{program}_0)$. Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}$. Properties 1, 2, and 4 hold. The set $\mathcal{O}(\mathcal{RI}_0)$

contains all the oracles that can be called at the beginning, and

$$\mathcal{Q}_2 = \bigcup_{2 \leq a \leq N_{\mathsf{rand+calls}}} \{Q_{loop}\{a/i'\}\} \cup \mathrm{reduce}(Q_0)\,,$$

so Property 3 holds. As mentioned in Section 6.1.6, the initial program $program_0$ does not contain locations in S_l, so Property 5 holds. As also mentioned in Section 6.1.6, $program_0$ contains no closure, and as mentioned in Chapter 7, $program_0$ contains no tagged function, no return, and no event except in parts $program(\mu_{\mathsf{role}})$ inside addthread. So Property 6(b) holds. By Assumption 7.1, Property 7 holds. The global store $globalstore_0$ maps each $l \in S_g$ to its initial value $initval_l$ and $globalstore(E_2, \mathcal{T}_0)$ maps each $f \in S_{priv}$ to its initial value $initval_f$ (the empty string "" when $(x[\,], f) \in \mathsf{files}$ and the empty list $[\,]$ when $(Tbl, f) \in \mathsf{tables}$), so Properties 8, 9, and 10 hold. The module set $\mathbb{G}_{\mathsf{get},\mathcal{MI}}(Q_0)$ and the role set $\mathcal{RI}_0$ correspond by definition of $\mathcal{RI}_0$, so Property 11 holds. The event lists are empty on both sides, so Property 12 holds. The sets $\mathcal{O}^\infty(\mathcal{I})$ with $\mathcal{I} \stackrel{\text{def}}{=} \emptyset$ and $\mathcal{O}_{\mathsf{call}}(\mathcal{CS}_0)$ are both empty, so Property 13 holds. The sets $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th_1^s)$, $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th_1^s))$, and $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th_1^s))$ where Th_1^s is the current thread of $\mathcal{CS}_0$ are also empty, so Property 14 holds. We have $0 + N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property 15 holds. We have $\alpha = 1$, so Property 16 holds. The set $\mathcal{I}$ is empty, so Property 17 holds. For all $\mathsf{role}\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{RI}_0$, we have $a' = 1$, so Property 18 holds. Therefore, $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}$. □

The following two lemmas serve to prove Property 4 of the invariant.

Lemma B.1 *If* $E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E} \rightarrow_p E', P', \mathcal{Q}', \mathcal{T}', \mathcal{R}', \mathcal{E}'$, *and* $\mathsf{fv}(P) \cup \mathsf{fv}(\mathcal{Q}) \cup \mathsf{fv}(\mathcal{R}) \subseteq \mathrm{Dom}(E)$, *then* $\mathsf{fv}(P') \cup \mathsf{fv}(\mathcal{Q}') \cup \mathsf{fv}(\mathcal{R}') \subseteq \mathrm{Dom}(E')$.

Proof This result is easily proved by cases on the applied reduction rule. □

We denote by $\mathfrak{C}^{\mathsf{cs}} = E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, rest$ an extended CryptoVerif configuration in which $rest$ is either nothing or $steps, \mathcal{CS}$.

Lemma B.2 *If* $E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, rest \rightsquigarrow_p E', P', \mathcal{Q}', \mathcal{T}', \mathcal{R}', \mathcal{E}', rest'$ *and* $\mathsf{fv}(P) \cup \mathsf{fv}(\mathcal{Q}) \cup \mathsf{fv}(\mathcal{R}) \subseteq \mathrm{Dom}(E)$, *then* $\mathsf{fv}(P') \cup \mathsf{fv}(\mathcal{Q}') \cup \mathsf{fv}(\mathcal{R}') \subseteq \mathrm{Dom}(E')$.

Proof This result is easily proved by cases on the applied reduction rule. By Lemma B.1, the rule (CryptoVerif) preserves the invariant. The rules (Enter Simulator) and (Simulator) leave the environment and the set of free variables unchanged. The rule (Leave Simulator) introduces a new free variable and adds it to the environment. □

The following lemma shows that a correct closure always remains correct during execution.

Lemma B.3 *Suppose that* $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$, $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}, \mathcal{I}$, $\mathcal{Q}_0' \leftrightarrow \mathcal{RI}', \mathcal{I}'$, R *is an oracle reference of the form* $O'[\widetilde{a'}]$ *when oracle* O' *is not under replication and* $O'[\_, \widetilde{a''}]$ *when* O' *is under replication, and one of the following two situations occurs:*

1. $E' \supseteq E$, $\mathcal{I}' = \mathcal{I} - \{O[\widetilde{a}]\}$, $\mathcal{Q}'_0 \supseteq \mathcal{Q}_0 \setminus \{\mathcal{Q}_0(O[\widetilde{a}])\}$, $\tau'_{\mathsf{O}} = \tau_{\mathsf{O}}$, *and* l'_{tok} *is a restriction of* l_{tok} *such that, if* $R = O'[\widetilde{a}']$, *then* $R \in \mathrm{Dom}(l'_{\mathsf{tok}})$.

2. $E' \supseteq E$, $\mathcal{I}' \supseteq \mathcal{I}$, $\mathcal{Q}'_0 \supseteq \mathcal{Q}_0$, $l'_{\mathsf{tok}} \supseteq l_{\mathsf{tok}}$, $\tau'_{\mathsf{O}} \supseteq \tau_{\mathsf{O}}$, *if* $R = O'[\widetilde{a'}]$, *then* $O'[\widetilde{a'}] \notin \mathcal{I}' \setminus \mathcal{I}$ *and* $O'[\widetilde{a'}] \in \mathrm{Dom}(l_{\mathsf{tok}})$, *and if* $R = O'[\_, \widetilde{a''}]$, *then for all* a, $O'\big[[a, +\infty[, \widetilde{a''}\big] \notin \mathcal{I}' \setminus \mathcal{I}$ *and* $O'[\_, \widetilde{a''}] \in \mathrm{Dom}(\tau_{\mathsf{O}})$.

Then $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

Proof Let us consider the first situation.

- Case $R = O[\widetilde{a}]$. Oracle O is not under replication. Since $\mathcal{I} - \{O[\widetilde{a}]\}$ is defined, we have $O[\widetilde{a}] \in \mathcal{I}$, and since $\mathcal{I}' = \mathcal{I} - \{O[\widetilde{a}]\}$, we have $O[\widetilde{a}] \notin \mathcal{I}'$. We also have $l'_{\mathsf{tok}}(O[\widetilde{a}]) = l_{\mathsf{tok}}(O[\widetilde{a}])$. So, by definition of $\mathsf{correctclosure}$,

$$\begin{aligned}
&\mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \\
&\quad = \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{false}}(Q)] \mid \\
&\qquad \text{for any } Q, env(token) = l'_{\mathsf{tok}}(O[\widetilde{a}])\} \\
&\quad \supseteq \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{false}}(\mathcal{Q}(O[\widetilde{a}]))] \mid \\
&\qquad env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}(O[\widetilde{a}])), env(token) = l_{\mathsf{tok}}(O[\widetilde{a}])\} \\
&\quad \supseteq \mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,.
\end{aligned}$$

- Case $R = O[\_, \widetilde{a''}]$ where $\widetilde{a} = a', \widetilde{a''}$ for some a'. Oracle O is under replication. Since $\mathcal{I} - \{O[\widetilde{a}]\}$ is defined, $O\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$, and since $\mathcal{I}' = \mathcal{I} - \{O[\widetilde{a}]\}$, we have $O\big[[a'+1, +\infty[, \widetilde{a''}\big] \in \mathcal{I}'$.

  Suppose that $a' < N_O$. Since $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}, \mathcal{I}$, there exist Q and i such that i does not occur in $\mathsf{fv}(Q)$, $\mathcal{Q}_0(O[a', \widetilde{a''}]) = Q\{a'/i\}$, and $\mathcal{Q}_0(O[a'+1, \widetilde{a''}]) = Q\{a'+1/i\}$. It is easy to see that $pm_{\mathrm{true}}(Q\{a'+1/i\}) = pm_{\mathrm{true}}(Q\{a'/i\})$, since the translation into OCaml does not depend on the indices. Moreover, $\mathsf{fv}(Q\{a'+1/i\}) = \mathsf{fv}(Q\{a'/i\})$ since i does not occur in $\mathsf{fv}(Q)$, so $env(E, Q\{a'+1/i\}) = env(E, Q\{a'/i\})$. Since $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$ and E' is an extension of E, we have $env(E', Q\{a'+1/i\}) = env(E, Q\{a'+1/i\})$. So, by definition of $\mathsf{correctclosure}$,

$$\begin{aligned}
&\mathsf{correctclosure}(O[_, \widetilde{a''}], \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \\
&\quad = \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(\mathcal{Q}'_0(O[a'+1, \widetilde{a''}]))] \mid \\
&\qquad \tau = \tau'_{\mathsf{O}}(O[_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E', \mathcal{Q}'_0(O[a'+1, \widetilde{a''}]))\} \\
&\quad = \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(\mathcal{Q}_0(O[a', \widetilde{a''}]))] \mid \\
&\qquad \tau = \tau_{\mathsf{O}}(O[_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}_0(O[a', \widetilde{a''}]))\} \\
&\quad = \mathsf{correctclosure}(O[_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,.
\end{aligned}$$

Suppose that $a' = N_O$. By definition of $\mathsf{correctclosure}$,

$$\begin{aligned}
&\mathsf{correctclosure}(O[_, \widetilde{a''}], \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \\
&\quad = \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(Q)] \mid \tau = \tau'_{\mathsf{O}}(O[_, \widetilde{a''}]), \text{for any } Q, env\} \\
&\quad \supseteq \{\mathsf{tagfunction}^{O,\tau}[env, pm_{\mathrm{true}}(\mathcal{Q}_0(O[a', \widetilde{a''}]))] \mid \\
&\qquad \tau = \tau_{\mathsf{O}}(O[_, \widetilde{a''}]), env \supseteq env_{\mathsf{prim}} \cup env(E, \mathcal{Q}_0(O[a', \widetilde{a''}]))\} \\
&\quad \supseteq \mathsf{correctclosure}(O[_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,.
\end{aligned}$$

Suppose that $a' > N_O$. We have $\mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(O[\_, \widetilde{a''}], \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$ since $E, \mathcal{Q}_0, l_{\mathsf{tok}}$ are not used and $\tau'_{\mathsf{O}} = \tau_{\mathsf{O}}$.

- Other cases. All references to $\mathcal{Q}'_0(O'[\widetilde{a'}])$ in the definition of $\mathsf{correctclosure}$ satisfy $O'[\widetilde{a'}] \neq O[\widetilde{a}]$. In this case, we have $\mathcal{Q}'_0(O'[\widetilde{a'}]) = \mathcal{Q}_0(O'[\widetilde{a'}])$. Since $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$ and E' is an extension of E, we have $env(E', \mathcal{Q}'_0(O'[\widetilde{a'}])) = env(E', \mathcal{Q}_0(O'[\widetilde{a'}])) = env(E, \mathcal{Q}_0(O'[\widetilde{a'}]))$. Moreover, when $R = O'[\widetilde{a'}]$, we have $l'_{\mathsf{tok}}(O'[\widetilde{a'}]) = l_{\mathsf{tok}}(O'[\widetilde{a'}])$. Hence, by definition of $\mathsf{correctclosure}$, $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

Let us now consider the second situation.

- Case $R = O'[\widetilde{a'}]$. Since $O'[\widetilde{a'}] \notin \mathcal{I}' \setminus \mathcal{I}$, we have $O'[\widetilde{a'}] \in \mathcal{I}'$ if and only if $O'[\widetilde{a'}] \in \mathcal{I}$. We have $l'_{\mathsf{tok}}(O'[\widetilde{a'}]) = l_{\mathsf{tok}}(O'[\widetilde{a'}])$.

  If $O'[\widetilde{a'}] \notin \mathcal{I}$, these points are sufficient to conclude that $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

  If $O'[\widetilde{a'}] \in \mathcal{I}$, there is an oracle $O'[\widetilde{a'}]$ in $\mathcal{Q}_0$; since $\mathcal{Q}'_0 \supseteq \mathcal{Q}_0$, we have $\mathcal{Q}'_0(O'[\widetilde{a'}]) = \mathcal{Q}_0(O'[\widetilde{a'}])$. Since $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$ and E' is an extension of E, we have $env(E', \mathcal{Q}'_0(O'[\widetilde{a'}])) = env(E', \mathcal{Q}_0(O'[\widetilde{a'}])) = env(E, \mathcal{Q}_0(O'[\widetilde{a'}]))$. So $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

- Case $R = O'[\_, \widetilde{a''}]$. Since for all a, $O'\big[[a, +\infty[, \widetilde{a''}\big] \notin \mathcal{I}' \setminus \mathcal{I}$, we have $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}'$ if and only if $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$.

  If there is no a' such that $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$, this point is sufficient to conclude that $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

  If $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$ and $a' \leq N_{O'}$, then there is an oracle $O'[a', \widetilde{a''}]$ in $\mathcal{Q}_0$; since $\mathcal{Q}'_0 \supseteq \mathcal{Q}_0$, we have $\mathcal{Q}'_0(O'[a', \widetilde{a''}]) = \mathcal{Q}_0(O'[a', \widetilde{a''}])$. Since $\mathsf{fv}(\mathcal{Q}_0) \subseteq \mathrm{Dom}(E)$ and E' is an extension of E, we obtain $env(E', \mathcal{Q}'_0(O'[a', \widetilde{a''}])) = env(E', \mathcal{Q}_0(O'[a', \widetilde{a''}])) = env(E, \mathcal{Q}_0(O'[a', \widetilde{a''}]))$. Moreover, we have $\tau'_{\mathsf{O}}(O'[\_, \widetilde{a''}]) = \tau_{\mathsf{O}}(O'[\_, \widetilde{a''}])$. So $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

  If $O'\big[[a', +\infty[, \widetilde{a''}\big] \in \mathcal{I}$ and $a' > N_{O'}$, then we have $\tau'_{\mathsf{O}}(O'[\_, \widetilde{a''}]) = \tau_{\mathsf{O}}(O'[\_, \widetilde{a''}])$, so $\mathsf{correctclosure}(R, \mathcal{I}', E', \mathcal{Q}'_0, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) = \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}_0, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. □

Let P'_{loop} be the process from line 8 to line 20 of Figure 8.2. Let P^j_{loop} be the process from line 13 to line 15 for the $\mathsf{if}\ o = o_j$ branch. Let P^R_{loop} be the process from line 19 to line 20. The expansion of the let construct with pattern-matching introduces a fresh variable. Let us denote $xs[i']$ the variable created for the let matching on line 7, $xa_j[i']$ and $xi_j[i']$ the variables created on lines 11 and 12 for oracle number j.

Proof (of Lemma 8.34) Let us consider $\mathfrak{C}^{\mathsf{cs}}$ and $\mathcal{CT}$ such that $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. Let $\mathfrak{C}^{\mathsf{cs}} = E, P_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E}, steps, \mathcal{CS}$ and let $\mathcal{C}$ be the last configuration of $\mathcal{CT}$. Let

$$\begin{aligned}
\mathcal{CS} &= ([Th_1, \ldots, Th_n], globalstore^s, tj), \mathcal{RI}, \mathcal{I}\,, \\
\mathcal{C} &= [Th'_1, \ldots, Th'_n], globalstore^o, tj, \mathcal{MI}, events\,, \\
Th_{tj} &= Th^s = \langle env^s, pe^s, stack^s, store^s\rangle\,, \\
Th'_{tj} &= Th^o = \langle env^o, pe^o, stack^o, store^o\rangle\,.
\end{aligned}$$

We use the exponent s for the elements of the simulator configuration and the exponent o for the elements of the OCaml configuration.

Let us first distinguish cases depending on whether Property 6(a) or Property 6(b) is satisfied for the current thread.

Case 1. Property 6(a) is satisfied for the current thread, that is, we are at the beginning of the initialization of a protocol thread. There exists a program $program'$ such that

$$\begin{aligned}
Th^s = \langle \emptyset, program_{\mathsf{prim}};; program'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots;; program'(\mathsf{role}_m[\widetilde{a_m}]);; \\
program', [\,], \emptyset\rangle\,.
\end{aligned}$$

There is no closure, no tagged function $\mathsf{tagfunction}^t\ pm$, no event, and no return in $program'$, except in $program(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$. The OCaml thread verifies $Th^o = \mathsf{replaceinitpm}(Th^s)$, so

$$Th^o = \langle \emptyset, program_{\mathsf{prim}};; program(\mu_{\mathsf{role}_1});; \ldots;; program(\mu_{\mathsf{role}_m});; program', [\,], \emptyset\rangle\,.$$

By Assumption 8.2, there is exactly one complete thread trace $\mathcal{TT}$ that begins at $\langle \emptyset, program_{\mathsf{prim}};;, [\,], \emptyset\rangle$, and the last thread of this trace is $\langle env_{\mathsf{prim}}, \varepsilon, [\,], \emptyset\rangle$. So there is no call to the random function inside the initialization of the primitives. Let $\mathcal{TT}(definitions)$ be the trace $\mathcal{TT}$ where, in each thread, we replace the empty definition list ε by *definitions*. As no OCaml reduction rule depends on the contents of a definition list, the trace $\mathcal{TT}(definitions)$ is a valid trace for any definition list *definitions*. So, by taking *definitions* the definitions after $program_{\mathsf{prim}}$ in Th^s and Th^o,

$$\begin{aligned}
Th^s \rightarrow^* Th^s_e &\stackrel{\text{def}}{=} \langle env_{\mathsf{prim}}, program'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots;; program'(\mathsf{role}_m[\widetilde{a_m}]);; \\
&\qquad program', [\,], \emptyset\rangle \\
Th^o \rightarrow^* Th^o_e &\stackrel{\text{def}}{=} \langle env_{\mathsf{prim}}, program(\mu_{\mathsf{role}_1});; \ldots;; program(\mu_{\mathsf{role}_m});; program', [\,], \emptyset\rangle
\end{aligned}$$

in exactly the same number of steps.

Let l_j ($j \leq m$) be m distinct locations. For $j \leq m+1$, let $Th^s_j \stackrel{\text{def}}{=} \langle env^s_j, program_j, [\,], store^s_j \rangle$ where $env^s_j \stackrel{\text{def}}{=} env_{\mathsf{prim}} \cup \{\mu_{\mathsf{role}_i}.\mathsf{init} \mapsto c_i \mid i < j\}$ with $c_i \stackrel{\text{def}}{=} \mathsf{tagfunction}^{\mathsf{role}_i,\tau_i}[env_i, pm'_{\mathsf{role}_i[\widetilde{a_i}]}]$ and $env_i \stackrel{\text{def}}{=} env^s_i[token \mapsto l_i]$,

$$program_j \stackrel{\text{def}}{=} program'(\mathsf{role}_j[\widetilde{a_j}]);; \ldots ;; program'(\mathsf{role}_m[\widetilde{a_m}]);; program' \text{ for } j \leq m$$

and $program_j \stackrel{\text{def}}{=} program'$ for $j = m+1$, and $store^s_j \stackrel{\text{def}}{=} \{l_i \mapsto \text{true} \mid i < j\}$. For $j \leq m$, the thread Th^s_j reduces as follows:

$$\begin{aligned}
Th^s_j &= \langle env^s_j, \mathsf{let}\ \mu_{\mathsf{role}_j}.\mathsf{init} = e^s_j;; program_{j+1}, [\,], store^s_j \rangle \\
&\quad \text{where } e^s_j \stackrel{\text{def}}{=} \mathsf{let}\ token = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}_j}\ pm'_{\mathsf{role}_j[\widetilde{a_j}]} \\
&\to \langle env^s_j, e^s_j, stack^s_j, store^s_j \rangle \\
&\quad \text{where } stack^s_j \stackrel{\text{def}}{=} [env^s_j, \mathsf{let}\ \mu_{\mathsf{role}_j}.\mathsf{init} = [\cdot];; program_{j+1}] \\
&\to^* \langle env_j, \mathsf{tagfunction}^{\mathsf{role}_j}\ pm'_{\mathsf{role}_j[\widetilde{a_j}]}, stack^s_j, store^s_{j+1} \rangle \\
&\quad \text{since } env_j = env^s_j[token \mapsto l_j] \text{ and } store^s_{j+1} = store^s_j[l_j \mapsto \text{true}] \\
&\to \langle env_j, c_j, stack^s_j, store^s_{j+1} \rangle \\
\to^* Th^s_{j+1} &= \langle env^s_{j+1}, program_{j+1}, [\,], store^s_{j+1} \rangle \\
&\quad \text{since } env^s_{j+1} = env^s_j[\mu_{\mathsf{role}_j}.\mathsf{init} \mapsto c_j]
\end{aligned}$$

Let $Th^o_j \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s_j)$. We have $Th^o_j = \langle env^o_j, program'_j, [\,], store^s_j \rangle$ where env^o_j is the environment env^s_j in which we replace $pm'_{\mathsf{role}_i[\widetilde{a_i}]}$ with $pm_{\mu_{\mathsf{role}_i}}$ for all $i < j$, $program'_j \stackrel{\text{def}}{=} program(\mu_{\mathsf{role}_j});; \ldots ;; program(\mu_{\mathsf{role}_m});; program'$ for $j \leq m$, and $program'_j \stackrel{\text{def}}{=} program'$ for $j = m+1$. For $j \leq m$, the thread Th^o_j reduces as follows:

$$\begin{aligned}
Th^o_j &= \langle env^o_j, \mathsf{let}\ \mu_{\mathsf{role}_j}.\mathsf{init} = e^o_j;; program'_{j+1}, [\,], store^s_j \rangle \\
&\quad \text{where } e^o_j \stackrel{\text{def}}{=} \mathsf{let}\ token = \mathsf{ref}\ \text{true}\ \mathsf{in}\ \mathsf{tagfunction}^{\mathsf{role}_j}\ pm_{\mu_{\mathsf{role}_j}} \\
&\to \langle env^o_j, e^o_j, stack^o_j, store^s_j \rangle \\
&\quad \text{where } stack^o_j \stackrel{\text{def}}{=} [env^o_j, \mathsf{let}\ \mu_{\mathsf{role}_j}.\mathsf{init} = [\cdot];; program'_{j+1}] \\
&\to^* \langle env'_j, \mathsf{tagfunction}^{\mathsf{role}_j}\ pm_{\mu_{\mathsf{role}_j}}, stack^o_j, store^s_{j+1} \rangle \\
&\quad \text{where } env'_j \stackrel{\text{def}}{=} env^o_j[token \mapsto l_j] \\
&\to \langle env'_j, c'_j, stack^o_j, store^s_{j+1} \rangle \\
&\quad \text{where } c'_j \stackrel{\text{def}}{=} \mathsf{tagfunction}^{\mathsf{role}_j,\tau_j}[env'_j, pm_{\mu_{\mathsf{role}_j}}] \\
\to^* Th^o_{j+1} &= \langle env^o_{j+1}, program'_{j+1}, [\,], store^s_{j+1} \rangle \\
&\quad \text{since } env^o_{j+1} = env^o_j[\mu_{\mathsf{role}_j}.\mathsf{init} \mapsto c'_j]
\end{aligned}$$

Moreover, $Th^s_e = Th^s_1$ and $Th^o_e = Th^o_1$, so $Th^s \to^* Th^s_{m+1}$ and $Th^o \to^* Th^o_{m+1}$. There are exactly the same number of steps in both traces. Let $steps^s$ be this number of steps.

Let $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}[\mathsf{Th} \mapsto Th^o_{m+1}]$. Since $Th^s \to^* Th^s_{m+1}$ without using (Random), we have $\mathcal{CS} \to^* \mathcal{CS}[\mathsf{Th} \mapsto Th^s_{m+1}]$

by (Globalstore1), (Toplevel), and (Simulator toplevel). Furthermore, by definition of N_{steps}, all traces of the OCaml program have at most N_{steps} steps, so in particular $|\mathcal{CT}_1| = |\mathcal{CT}| + steps^s \leq N_{\mathsf{steps}}$. Hence, by Property 15, $steps \geq N_{\mathsf{steps}} - |\mathcal{CT}| \geq steps^s$. So, with $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - steps^s, \mathcal{CS}[\mathsf{Th} \mapsto Th^s_{m+1}]$, we have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathfrak{C}_1^{\mathsf{cs}}$ by (Simulator) since *steps* remains positive during the reduction. (More generally, the same reasoning shows that, if the simulator trace has at most as many steps as the OCaml trace, then the extended CryptoVerif configuration can reduce by (Simulator) because *steps* remains positive by Property 15. We shall not detail this point in the other cases.)

Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. Properties 1, 2, 3, 4, 7, 8, 9, 10, 11, 12 are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. As mentioned in Section 6.1.6, there are no local store locations in the initial program, so there are no local store locations in *program*′, so the locations $l_1, \ldots, l_m$ are the only local store locations present in Th^s_{m+1}, and they are all in the domain of $store^s_{m+1}$. So Property 5 holds. Let l_{tok} be the empty function. The set $\mathcal{O}_{\mathsf{call}}(Th^s_{m+1})$ is empty. We have that $\mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s_{m+1}), l_{\mathsf{tok}}) = \emptyset$ and $Th^o_{m+1} = \mathsf{replaceinitpm}(Th^s_{m+1}) \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s_{m+1}), \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so Property 6(b)i holds for the current thread. Using the function $l_{\mathsf{init\text{-}tok}}$ that maps $\mathsf{role}_j[\widetilde{a_j}]$ to l_j for $j \leq m$, Property 6(b)ii holds for the current thread. The environment of the tagged closures that we created contains env_{prim}, so Property 6(b)iii holds for the current thread. Since there is no tagged function, no event and no return in *program*′ except in $program(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$, Property 6(b)iv holds for the current thread. Threads that are not the current thread did not change, so Property 6 holds. The only change in the oracle sets is that the roles $\mathsf{role}_j[\widetilde{a_j}]$ are transferred from $\mathcal{R}_{\mathsf{init\text{-}function}}(Th^s)$ to $\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s)$, so Properties 13 and 14 are preserved. We have

$$|\mathcal{CT}_1| + steps - steps^s = |\mathcal{CT}| + steps^s + steps - steps^s \geq N_{\mathsf{steps}}$$

so Property 15 holds. Properties 16, 17, and 18 are preserved, because all components of these inequalities are unchanged. Therefore, we have proved that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$.

Case 2. Property 6(b) is satisfied for the current thread. We now distinguish cases on the form of the simulator configuration $\mathcal{CS}$.

Case 2.1. The current expression of $\mathcal{CS}$ is $pe^s = \mathsf{call}(O_j[\widetilde{a}])\ (v_1, \ldots, v_{m_j})$ and $\mathcal{CS}$ cannot reduce, that is, the configuration $\mathcal{CS}$ makes a successful call to $O_j[\widetilde{a}]$, an oracle not under replication. By definition of *simreturn*, $simreturn(\mathcal{CS}) \stackrel{\text{def}}{=} (\mathsf{repr}(\mathcal{CS}), o_j, \widetilde{a}, args)$ where $args \stackrel{\text{def}}{=} (b_1, \ldots, b_{m_j})$ and $b_k \stackrel{\text{def}}{=} \mathbb{G}^{-1}_{\mathsf{val}T_{j,k}}(v_k)$ for $k \leq m_j$.

So $\mathfrak{C}^{\mathsf{cs}}$ reduces in several steps into the configuration $E_1, P'_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E}$ that corresponds to line 8 where

$$\begin{aligned} E_1 \stackrel{\text{def}}{=} E[&xs[\alpha] \mapsto (\mathsf{repr}(\mathcal{CS}), o_j, \widetilde{a}, args), \\ &s'[\alpha] \mapsto \mathsf{repr}(\mathcal{CS}), o[\alpha] \mapsto o_j, i[\alpha] \mapsto \widetilde{a}, args[\alpha] \mapsto args]\,. \end{aligned}$$

Let $a'_1, \ldots, a'_{n_j} \stackrel{\text{def}}{=} \widetilde{a}$. As $E_1(o[\alpha]) = o_j$, this configuration reduces in several steps

into the configuration $E_2, P^j_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E}$ where

$$E_2 \stackrel{\text{def}}{=} E_1[xa_j[\alpha] \mapsto args, a_{j,1}[\alpha] \mapsto b_1, \ldots, a_{j,m_j}[\alpha] \mapsto b_{m_j}, \\ xi_j[\alpha] \mapsto \widetilde{a}, i_{j,1}[\alpha] \mapsto a'_1, \ldots, i_{j,n_j}[\alpha] \mapsto a'_{n_j}].$$

The oracle $O_j[\widetilde{a}]$ is in $\mathcal{I}$, otherwise $\mathcal{CS}$ could reduce using (FailedCall1). By Property 3 of the invariant, there exists $\mathcal{Q}_0$ such that $\mathcal{Q}_0 \leftrightarrow \mathcal{I}, \mathcal{RI}$ and $\mathcal{Q} = \{Q_{loop}\{a/i'\} \mid \alpha < a \leq N_{\mathsf{rand+calls}}\} \cup \mathcal{Q}_0$. Let $Q \stackrel{\text{def}}{=} \mathcal{Q}_0(O_j[\widetilde{a}])$. The oracle definition Q is of the form $O_j[\widetilde{a}](x_1[\widetilde{a}] : T_{j,1}, \ldots, x_{m_j}[\widetilde{a}] : T_{j,m_j}) := P_O$. The previous configuration reduces in one step into $\mathfrak{C} \stackrel{\text{def}}{=} E_3, P_O, \mathcal{T}, \mathcal{Q}_1, \mathcal{R}_1, \mathcal{E}$ where

$$E_3 \stackrel{\text{def}}{=} E_2[x_1[\widetilde{a}] \mapsto b_1, \ldots, x_{m_j}[\widetilde{a}] \mapsto b_{m_j}]$$
$$\mathcal{R}_1 \stackrel{\text{def}}{=} ((r_{j,1}, \ldots, r_{j,m'_j}), \mathsf{return}(simulate'_{O_j}(s', (r_{j,1}, \ldots, r_{j,m'_j})), \text{true}), \\ \mathsf{return}(simulate''_{O_j}(s'), \text{true})) :: \mathcal{R}_{loop}(\alpha)$$
$$\mathcal{Q}_1 \stackrel{\text{def}}{=} \mathcal{Q} \setminus \{Q\}$$

Let us now look at $\mathcal{C}$. By the invariant, there exists an injection l_{tok} that satisfies Property 6(b)i. The current expression pe^o is of the form $c\ (v_1, \ldots, v_{m_j})$, where $c \in \mathsf{correctclosure}(O_j[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, that is, $c = \mathsf{tagfunction}^{O_j,\tau}[env^o_1, pm_{\text{false}}(Q)]$ where $env^o_1 \supseteq env_{\mathsf{prim}} \cup env(E, Q)$ and $env^o_1(token) = l_{\mathsf{tok}}(O_j[\widetilde{a}])$. By the same property, $store^o(l_{\mathsf{tok}}(O_j[\widetilde{a}])) = \text{true}$.

$$Th^o = \langle env^o, c\ (v_1, \ldots, v_{m_j}), stack^o, store^o\rangle \\ \to \langle env^o_1, \mathsf{match}\ (v_1, \ldots, v_{m_j})\ \mathsf{with}\ pm_{\text{false}}(Q), stack^o, store^o\rangle$$
$$\to Th^o_1 \stackrel{\text{def}}{=} \langle env^o_2, e, stack^o, store^o\rangle$$
$$\text{where } env^o_2 \stackrel{\text{def}}{=} env^o_1[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto v_1, \ldots, \mathbb{G}_{\mathsf{var}}(x_{m_j}) \mapsto v_{m_j}] \text{ and}$$
$$e \stackrel{\text{def}}{=} \mathsf{if}\ (!token)\ \&\& \\ (\mathbb{G}_{\mathsf{pred}}(T_{j,1})\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \ldots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_{j,m_j})\ \mathbb{G}_{\mathsf{var}}(x_{m_j})) \\ \mathsf{then}\ (token := \text{false}; e')\ \mathsf{else\ raise\ Bad\_Call}$$
$$e' \stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{file}}(x_1[\widetilde{a}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_{m_j}[\widetilde{a}]); \mathbb{G}(P_O)$$

For all $k \leq m_j$, there exists b_k such that $\mathbb{G}_{\mathsf{val}T_{j,k}}(b_k) = v_k$, so $\mathbb{G}_{\mathsf{pred}}(T_{j,k})\ \mathbb{G}_{\mathsf{var}}(x_k)$ evaluates to true using Proposition 8.5. Moreover, $env^o_2(token) = l_{\mathsf{tok}}(O_j[\widetilde{a}])$. So, the configuration $\mathcal{C}$ reduces as follows

$$\mathcal{C} \to^* \mathcal{C}' \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto Th^o_1] \\ \to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env^o_2, token := \text{false}; e', stack^o, store^o_1\rangle] \\ \to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env^o_2, e', stack^o, store^o_2\rangle] \\ \text{where } store^o_2 \stackrel{\text{def}}{=} store^o_1[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \text{false}]$$
$$\to^* \mathcal{C}_1 \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env^o_2, \mathbb{G}(P_O), stack^o, store^o_3\rangle, \mathsf{globalstore} \mapsto globalstore^o_1] \\ \text{by Lemma A.1 applied } m_j \text{ times}$$

where $store^o_3 \supseteq store^o_2$, $store^o_1 \supseteq store^o$, $globalstore^o_1 \supseteq globalstore(E_3, \mathcal{T})$ since $globalstore^o \supseteq globalstore(E, \mathcal{T})$ by Property 9 of the invariant, and $globalstore^o_1(l) = globalstore^o(l)$ for all $l \notin S_{priv}$.

We prove that for any traces $\mathfrak{CT}_1, \ldots, \mathfrak{CT}_m$ beginning at $\mathfrak{C}$ such that $\sum_{i \leq m} \Pr[\mathfrak{CT}_i] = 1$, none of these traces is a prefix of another, and there is no intermediate configuration inside any of these traces with a return, end, call, or loop as current process, there exist m disjoint sets of OCaml traces $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_m$ all starting at $\mathcal{C}_1$ such that none of these traces is a prefix of another of these traces, $\Pr[\mathcal{CTS}_i] = \Pr[\mathfrak{CT}_i]$ for all $i \leq m$, and if $\mathcal{C}_4$ is the last configuration of a trace $\mathcal{CT}' \in \mathcal{CTS}_i$, then $\mathcal{C}_4 = \mathcal{C}[\mathsf{Th} \mapsto \mathit{Th}_4^o, \mathsf{globalstore} \mapsto \mathit{globalstore}_4^o, \mathsf{events} \mapsto \mathit{events}_4]$ where

$$\begin{aligned}
\mathit{Th}_4^o &= \langle \mathit{env}_4^o, \mathbb{G}(P_4), \mathit{stack}^o, \mathit{store}_4^o \rangle \text{ with} \\
&\quad \mathit{env}_4^o \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_4, P_4) \text{ and } \mathit{store}_4^o \supseteq \mathit{store}_3^o\,, \\
\mathit{globalstore}_4^o &\supseteq \mathit{globalstore}(E_4, \mathcal{T}_4)\,, \\
\mathit{globalstore}_4^o(l) &= \mathit{globalstore}^o(l) \text{ for all } l \notin S_{priv}\,, \\
\mathit{events}_4 &= \mathbb{G}_{\mathsf{ev}}(\mathcal{E}_4)\,,
\end{aligned}$$

and the last configuration of $\mathfrak{CT}_i$ is $E_4, P_4, \mathcal{T}_4, \mathcal{Q}_1, \mathcal{R}_1, \mathcal{E}_4$.

The proof proceeds by induction on the total length of the traces $\mathfrak{CT}_1, \ldots, \mathfrak{CT}_m$. In the base case, $m = 1$ and $\mathfrak{CT}_1$ is the trace that consists only of the configuration $\mathfrak{C}$. Let $\mathcal{CTS}_1$ consist of the single trace that contains just the configuration $\mathcal{C}_1$. We have $\mathit{env}_2^o \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_3, P_O)$ since $\mathit{env}^o \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E, Q)$, the variables $x_1[\widetilde{a}], \ldots, x_{m_j}[\widetilde{a}]$ are added on the CryptoVerif side, and $\mathbb{G}_{\mathsf{var}}(x_1), \ldots, \mathbb{G}_{\mathsf{var}}(x_{m_j})$ are added correspondingly on the OCaml side. As shown above, $\mathit{globalstore}_1^o \supseteq \mathit{globalstore}(E_3, \mathcal{T})$ and $\mathit{globalstore}_1^o(l) = \mathit{globalstore}^o(l)$ for all $l \notin S_{priv}$. By Property 12 of the invariant, $\mathit{events} = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$. So the property holds for the base case. The inductive case follows from Lemma 8.15.

Let us take the maximal CryptoVerif traces $\mathfrak{CT}_1, \ldots, \mathfrak{CT}_n$ that begin at $\mathfrak{C}$ and that contain no return, end, call, or loop as current process in intermediate configurations. Let $\mathcal{CTS}_1, \ldots, \mathcal{CTS}_n$ the trace sets as defined above. The final configurations of the CryptoVerif traces $\mathfrak{CT}_i$ contain either return or end, since the oracle $O_j[\widetilde{a}]$ does not contain loop or call constructs. Let us take one such trace $\mathfrak{CT}_i$ and a trace $\mathcal{CT}' \in \mathcal{CTS}_i$. Let $\mathfrak{C}_4$ and $\mathcal{C}_4$ be the last configurations of $\mathfrak{CT}_i$ and $\mathcal{CT}'$ respectively. Let $\mathfrak{C}_4 = E_4, P_4, \mathcal{T}_4, \mathcal{Q}_1, \mathcal{R}_1, \mathcal{E}_4$. We distinguish cases on the form of P_4.

- If $P_4 = \mathsf{end}$,

$$\begin{aligned}
\mathfrak{C}_4 &\rightsquigarrow E_4, \mathsf{return}(\mathit{simulate}''_{O_j}(s'[\alpha]), \mathrm{true}), \mathcal{T}_4, \mathcal{Q}_1, \mathcal{R}_{\mathit{loop}}(\alpha), \mathcal{E}_4 \\
&\rightsquigarrow E_5, P_{\mathit{return\text{-}loop}}(\alpha), \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4 \\
&\quad \text{where } E_5 \stackrel{\text{def}}{=} E_4[r'_{\alpha,r}[\,] \mapsto s, b_{\alpha,r} \mapsto \mathrm{true}], \\
&\qquad s \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{CS}'), \\
&\qquad \mathcal{CS}' \text{ is } \mathcal{CS} \text{ in which the current expression is} \\
&\qquad \text{replaced with } \mathsf{raise\ Match_failure} \text{ and the set } \mathcal{I} \\
&\qquad \text{is replaced with } \mathcal{I}' \stackrel{\text{def}}{=} \mathcal{I} - (O_j[\widetilde{a}]) \\
&\rightsquigarrow E_5, P_5, \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4 \\
&\quad \text{where } P_5 \stackrel{\text{def}}{=} \mathsf{let}\ r[\,] : T_{\mathcal{CS}} = \mathsf{loop}\ O_{\mathrm{loop}}[\alpha + 1](r'_{\alpha,r}[\,]) \\
&\qquad\qquad \mathsf{in\ end\ else\ end}
\end{aligned}$$

$\rightsquigarrow E_5, P_6, \mathcal{T}_4, \mathcal{Q}_1, [x[\,], \mathsf{return}(x[\,]), \mathsf{end}], \mathcal{E}_4$

where $P_6 \stackrel{\text{def}}{=} \mathsf{let}\ (r'_{\alpha+1,r}[\,] : T_{\mathcal{CS}}, b_{\alpha+1,r}[\,] : bool) =$
$O_{\text{loop}}[\alpha+1](r'_{\alpha,r})\ \mathsf{in}\ P_{return\text{-}loop}(\alpha+1)\ \mathsf{else\ end}$

$\rightsquigarrow E_6, P_{loop}\{\alpha+1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{R}_{loop}(\alpha+1), \mathcal{E}_4$

where $E_6 \stackrel{\text{def}}{=} E_5[s[\alpha+1] \mapsto s]$,
$\mathcal{Q}_2 \stackrel{\text{def}}{=} \mathcal{Q}_1 \setminus \{Q_{loop}\{\alpha+1/i'\}\}$

(Let $\mathcal{CT}''$ be the trace $\mathcal{CT}$ followed by $\mathcal{CT}'$. By definition of $N_{\mathsf{rand+calls}}$, $N_{\mathsf{rand+calls}} \geq \left(N_{\mathsf{rand}}(\mathcal{CT}'') + \sum_{O,\tau} N_{\mathsf{calls}}(O,\tau,\mathcal{CT}'')\right) + 1 = \left(N_{\mathsf{rand}}(\mathcal{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O,\tau,\mathcal{CT})\right) + 2$ since $\mathcal{CT}''$ makes one more call to O_j than $\mathcal{CT}$. So, by Property 16, $N_{\mathsf{rand+calls}} \geq \alpha+1$. So, by Property 3, $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}$, so $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}_1$.)

$\rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E_6, P_{loop}\{\alpha+1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{R}_{loop}(\alpha+1), \mathcal{E}_4, N_{\mathsf{steps}}, \mathcal{CS}'$

By definition of the translation of end, the current expression of $\mathcal{C}_4$ is $\mathsf{raise\ Match\_failure}$. Let $\mathcal{CT}''$ be the trace $\mathcal{CT}$ followed by $\mathcal{CT}'$. The last configuration of $\mathcal{CT}''$ is $\mathcal{C}_4$.

Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$. By the form of $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{C}_4$, Properties 1 and 2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ where we removed the oracles $O_j[\widetilde{a}]$ and $O_{\text{loop}}[\alpha+1]$. We have $\mathcal{I}' = \mathcal{I} - (O_j[\widetilde{a}])$, so Property 3 is preserved. Property 4 is an immediate consequence of Lemma B.2. No new locations were created in the simulator, and the domains of stores can only grow, so Property 5 is preserved.

For all threads $tj' \neq tj$, the thread tj' does not change so, to prove Property 6, we just have to show that Property 6(b)i is preserved; the other elements of Property 6 are obviously preserved. Suppose that thread tj' satisfies Property 6(b)i initially, with a function l_{tok}. By Lemma B.3, Item 1, for all $\mathsf{call}(R)$ that occur in $Th''_{tj'} \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th_{tj'})$, we have $\mathsf{correctclosure}(R, \mathcal{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $\mathsf{replacecalls}(Th''_{tj'}, \mathcal{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathsf{replacecalls}(Th''_{tj'}, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Furthermore, the oracle $O_j[\widetilde{a}]$ is in $\mathcal{O}_{\mathsf{call}}(Th_{tj})$, so by Property 14 of the invariant, it is not in $\mathcal{O}_{\mathsf{call}}(Th_{tj'})$. Hence, $\mathcal{O}_{\mathsf{call}}(Th_{tj'}) \cap \mathcal{I}' = \mathcal{O}_{\mathsf{call}}(Th_{tj'}) \cap \mathcal{I}$ and $\mathcal{O}_{\mathsf{call}}(Th_{tj'}) \setminus \mathcal{I}' = \mathcal{O}_{\mathsf{call}}(Th_{tj'}) \setminus \mathcal{I}$, so $\mathsf{gettokens}(\mathcal{I}', \mathcal{O}_{\mathsf{call}}(Th_{tj'}), l_{\mathsf{tok}}) = \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th_{tj'}), l_{\mathsf{tok}})$. So thread tj' continues to verify Property 6(b)i with the same function l_{tok}.

Let us now consider the current thread. The current thread of the simulator is $Th_1^s = \langle env^s, \mathsf{raise\ Match\_failure}, stack^s, store^s \rangle$ and the current thread on the OCaml side is $Th_4^o = \langle env_4^o, \mathsf{raise\ Match\_failure}, stack^o, store_4^o \rangle$ where $store_4^o \supseteq store_3^o$. By Property 6(b)i, there exist $store_5^o$ and l_{tok} such that

$$\langle env^o, pe^o, stack^o, store_5^o \rangle \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s), \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$$
$$store_5^o \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}}) \subseteq store^o\,.$$

Let us denote $Th'' \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s)$ and $Th''_1 \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s_1)$. The thread Th''_1 is the thread Th'' in which the current expression is replaced with raise Match_failure. This is an exceptional value, so the definition of replacecalls allows any environment in the threads it returns, hence

$$\begin{aligned} Th^o_5 \stackrel{\text{def}}{=} \langle env^o_4, \mathsf{raise\ Match\_failure}, stack^o, store^o_5\rangle \\ \in \mathsf{replacecalls}(Th''_1, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})\,. \end{aligned}$$

Let $l'_{\mathsf{tok}} \stackrel{\text{def}}{=} l_{\mathsf{tok}|\mathcal{O}_{\mathsf{call}}(Th^s_1)}$. By Lemma B.3, Item 1, for all $\mathsf{call}(R)$ that occur in Th''_1, we have $\mathsf{correctclosure}(R, \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $Th^o_5 \in \mathsf{replacecalls}(Th''_1, \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau_{\mathsf{O}})$. We have

$$\begin{aligned} &store^o_5 \cup \mathsf{gettokens}(\mathcal{I}', \mathcal{O}_{\mathsf{call}}(Th^s_1), l'_{\mathsf{tok}}) \\ &\quad \subseteq store^o_5 \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}})[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathrm{false}] \\ &\quad \subseteq store^o[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathrm{false}] \\ &\quad \subseteq store^o_1[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \mathrm{false}] = store^o_2 \subseteq store^o_3 \subseteq store^o_4\,, \end{aligned}$$

so Property 6(b)i holds with the function l'_{tok}. Properties 6(b)ii, 6(b)iii, 6(b)iv are clearly preserved, so Property 6 holds.

Properties 7, 8, and 11 are also preserved. Properties 9, 10, and 12 hold because they are kept inside Lemma 8.15. We have $\mathcal{O}^\infty(\mathcal{I}') = \mathcal{O}^\infty(\mathcal{I}) \setminus \{O_j[\widetilde{a}]\}$. If there remains no occurrence of $\mathsf{call}(O_j[\widetilde{a}])$ in the thread Th^s_1, then

$$\begin{aligned} \mathcal{O}_{\mathsf{call}}(Th^s_1) = \mathcal{O}_{\mathsf{call}}(Th^s) \setminus \{O_j[\widetilde{a}]\} \text{ and} \\ \mathcal{O}_{\mathsf{call}}(\mathcal{CS}') = \mathcal{O}_{\mathsf{call}}(\mathcal{CS}) \setminus \{O_j[\widetilde{a}]\}\text{, so} \\ \mathcal{O}^\infty(\mathcal{I}') \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS}') = \mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS}) \setminus \{O_j[\widetilde{a}]\}\,. \end{aligned}$$

Otherwise,

$$\begin{aligned} \mathcal{O}_{\mathsf{call}}(Th^s_1) = \mathcal{O}_{\mathsf{call}}(Th^s) \text{ and} \\ \mathcal{O}_{\mathsf{call}}(\mathcal{CS}') = \mathcal{O}_{\mathsf{call}}(\mathcal{CS})\text{, so} \\ \mathcal{O}^\infty(\mathcal{I}') \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS}') = \mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})\,. \end{aligned}$$

We also have $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th^s_1) = \mathcal{O}_{\mathsf{call\text{-}repl}}(Th^s)$, $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th^s_1)) = \mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th^s))$, and $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s_1)) = \mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$, so Property 14 is preserved. The set $\mathcal{O}^\infty(\mathcal{I}') \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS}')$ is included in $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ and the set $\mathsf{willbeavailable}(\mathcal{CS}')$ is included in $\mathsf{willbeavailable}(\mathcal{CS})$, so Property 13 is preserved. We have $|\mathcal{CT}''| + N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property 15 holds. The number of calls to O_j increases by 1 and α increases by 1, so Property 16 is preserved. Properties 17 and 18 are preserved, because all components of these inequalities are unchanged. So $\mathfrak{C}^{\mathsf{cs}}_1 \equiv \mathcal{CT}''$ in this case.

- If $P_4 = \mathsf{return}(M_1, \ldots, M_{m'_j}); Q'$, let c_i $(i \leq m'_j)$ be the CryptoVerif values

such that $E_4, M_i \Downarrow c_i$.

$$\begin{aligned}
\mathfrak{C}_4 \rightsquigarrow\ & E_5, \mathsf{return}(\mathit{simulate}'_{O_j}(s'[\alpha], (r_{j,1}[\alpha], \ldots, r_{j,m'_j}[\alpha])), \mathrm{true}), \mathcal{T}_4, \mathcal{Q}_1, \\
& \mathcal{R}, \mathcal{E}_4 \\
& \text{where } E_5 \stackrel{\text{def}}{=} E_4[r_{j,1} \mapsto c_1, \ldots, r_{j,m'_j} \mapsto c_{m'_j}] \\
\rightsquigarrow^*\ \mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=}\ & E_6, P_{\mathit{loop}}\{\alpha + 1/i'\}, \mathcal{T}_4, \mathcal{Q}_2, \mathcal{R}_{\mathit{loop}}(\alpha+1), \mathcal{E}_4, N_{\mathsf{steps}}, \mathcal{CS}' \\
& \text{where } E_6 \supseteq E_5[s[\alpha+1] \mapsto \mathsf{repr}(\mathcal{CS}')], \\
& \qquad \mathcal{Q}_2 \stackrel{\text{def}}{=} \mathcal{Q}_1 \cup \mathrm{reduce}(Q') \setminus \{Q_{\mathit{loop}}\{\alpha + 1/i'\}\} \\
& \qquad \mathsf{repr}(\mathcal{CS}') = \mathit{simulate}'_{O_j}(\mathsf{repr}(\mathcal{CS}), (c_1, \ldots, c_{j,m'_j}))
\end{aligned}$$

where we show that $Q_{\mathit{loop}}\{\alpha + 1/i'\} \in \mathcal{Q}_1$ using Property 16 as in the case $P_4 = \mathsf{end}$.

Suppose that $\mathrm{reduce}'(Q') = [(Q_1, b_1), \ldots, (Q_l, b_l)]$ and $\mathsf{oracles}'(Q') = [O'_1[\widetilde{a_1}], \ldots, O'_l[\widetilde{a_l}]]$. A thread $\langle \mathit{env}, \mathbb{G}_{\mathsf{O}}(Q_i, b_i), \mathit{stack}, \mathit{store}\rangle$ where $\mathit{env} \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_4, P_4)$ reduces into $\langle \mathit{env}', c(Q_i, b_i), \mathit{stack}, \mathit{store}'\rangle$ where $c(Q_i, b_i) \stackrel{\text{def}}{=} \mathsf{tagfunction}^{O'_i, \tau_i}[\mathit{env}', \mathit{pm}_{b_i}(Q_i)]$ and

- if $b_i = \mathrm{false}$, then $\mathit{env}' = \mathit{env}[\mathit{token} \mapsto l_i]$ and $\mathit{store}' = \mathit{store}[l_i \mapsto \mathrm{true}]$ where l_i is a fresh location: $l_i \notin \mathrm{Dom}(\mathit{store})$;
- if $b_i = \mathrm{true}$, then $\mathit{env}' = \mathit{env}$ and $\mathit{store}' = \mathit{store}$.

So in both cases, $\mathit{env}' \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_4, P_4)$.

Let $\mathit{Th}_4^o = \langle \mathit{env}_4^o, \mathbb{G}(P_4), \mathit{stack}^o, \mathit{store}_4^o\rangle$ be the current thread of $\mathcal{C}_4$. We have $\mathit{env}_4^o \supseteq \mathit{env}_{\mathsf{prim}} \cup \mathit{env}(E_4, P_4)$ and $\mathit{store}_4^o \supseteq \mathit{store}_3^o$.

$$\begin{aligned}
\mathit{Th}_4^o =\ & \langle \mathit{env}_4^o, (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_l, b_l), \mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{m'_j})), \\
& \mathit{stack}^o, \mathit{store}_4^o\rangle \\
\rightarrow^*\ & \langle \mathit{env}_4^o, (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_l, b_l), \mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \\
& \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), \mathit{stack}^o, \mathit{store}_5^o\rangle \\
& \qquad\qquad \text{by Lemma 8.13 applied } m'_j \text{ times} \\
\rightarrow^*\ \mathit{Th}_5^o \stackrel{\text{def}}{=}\ & \langle \mathit{env}_4^o, (c(Q_1, b_1), \ldots, c(Q_l, b_l), \mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), \\
& \mathit{stack}^o, \mathit{store}_6^o\rangle
\end{aligned}$$

where $\mathit{store}_6^o \stackrel{\text{def}}{=} \mathit{store}_5^o \cup \{l_i \mapsto \mathrm{true} \mid b_i = \mathrm{false}\} \supseteq \mathit{store}_5^o \supseteq \mathit{store}_4^o$.

Let $\mathcal{CT}''$ be the trace $\mathcal{CT}$ followed by $\mathcal{CT}'$ and extended until $\mathcal{C}_5 \stackrel{\text{def}}{=} \mathcal{C}_4[\mathsf{Th} \mapsto \mathit{Th}_5^o]$. Let $\mathcal{I}'$ be the set $\mathcal{I}$ of $\mathcal{CS}'$.

Let us now prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$. We define $\tau'_{\mathsf{O}} \stackrel{\text{def}}{=} \tau_{\mathsf{O}} \cup \{O'_i[\_, \widetilde{a}] \mapsto \tau_i \mid b_i = \mathrm{true}\}$. (This function is well defined, because $O'_i[\_, \widetilde{a}] \notin \mathrm{Dom}(\tau_{\mathsf{O}})$. Indeed, for any a', $O'_i[a', \widetilde{a}] \in \mathsf{willbeavailable}(\mathcal{CS})$, so by Property 13, $O'_i[a', \widetilde{a}] \notin \mathcal{O}^\infty(\mathcal{I})$, hence for any a', $O'_i\big[[a', +\infty[, \widetilde{a}\big] \notin \mathcal{I}$. The main reason why we introduced the set $\mathcal{O}^\infty(\mathcal{I})$ is that at this point, we are able to distinguish between an oracle under replication that has not been called yet and

an oracle whose calls have been exhausted. If we used the set $\mathcal{O}(\mathcal{I})$ instead here, we would not be able to conclude that there is no oracle $O'_i\big[[a', +\infty[, \widetilde{a}\big]$ in $\mathcal{I}$: if $a' > N_{O'_i}$, then $\mathcal{O}(\{O'_i\big[[a', +\infty[, \widetilde{a}\big]\}) = \emptyset$.) By the form of $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{C}_5$, Properties 1 and 2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ where we removed the oracles $O_j[\widetilde{a}]$ and $O_{\text{loop}}[\alpha + 1]$ and where we added the new oracles reduce$'(Q')$. By definition of $simulate'_{O_j}$, the set $\mathcal{I}'$ is the set $\mathcal{I}$ where we removed $O_j[\widetilde{a}]$ and added the elements of $\mathsf{oracles}'(Q')$. So Property 3 is preserved. We also have $\mathcal{Q}_2(O'_i[\widetilde{a}_i]) = Q_i$ for $i \leq l$. Property 4 is an immediate consequence of Lemma B.2. No new locations were created in the simulator, and the domains of stores can only grow, so Property 5 is preserved.

For all threads $tj' \neq tj$, the thread tj' does not change so, to prove Property 6, we just have to show that Property 6(b)i is preserved; the other elements of Property 6 are obviously preserved. Suppose that thread tj' satisfies Property 6(b)i initially, with a function l_{tok}. By Lemma B.3, Items 1 and 2, for all $\mathsf{call}(R)$ that occur in $Th''_{tj'} \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th_{tj'})$, we have $\mathsf{correctclosure}(R, \mathcal{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, so $\mathsf{replacecalls}(Th''_{tj'}, \mathcal{I}', E_6, \mathcal{Q}_2, l_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{replacecalls}(Th''_{tj'}, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Furthermore, the oracle $O_j[\widetilde{a}]$ is in $\mathcal{O}_{\mathsf{call}}(Th_{tj})$ and the oracles $O'_i[\widetilde{a}_i]$ that are not under replication are in $\mathsf{willbeavailable}(\mathcal{CS})$, so by Properties 13 and 14 of the invariant, they are not in $\mathcal{O}_{\mathsf{call}}(Th_{tj'})$. Hence, $\mathcal{O}_{\mathsf{call}}(Th_{tj'}) \cap \mathcal{I}' = \mathcal{O}_{\mathsf{call}}(Th_{tj'}) \cap \mathcal{I}$ and $\mathcal{O}_{\mathsf{call}}(Th_{tj'}) \setminus \mathcal{I}' = \mathcal{O}_{\mathsf{call}}(Th_{tj'}) \setminus \mathcal{I}$, so $\mathsf{gettokens}(\mathcal{I}', \mathcal{O}_{\mathsf{call}}(Th_{tj'}), l_{\mathsf{tok}}) = \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th_{tj'}), l_{\mathsf{tok}})$. So thread tj' continues to verify Property 6(b)i with the same function l_{tok}.

Let us now consider the current thread. The current thread of the simulator is $Th_1^s = \langle env^s, pe_1^s, stack^s, store^s \rangle$ where $pe_1^s \stackrel{\text{def}}{=} (\mathsf{call}(O'_1[\widetilde{a}_1]), \ldots, \mathsf{call}(O'_l[\widetilde{a}_l]), \mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$ and the current thread on the OCaml side is $Th_5^o = \langle env_4^o, pe_6^o, stack^o, store_6^o \rangle$ where $pe_6^o \stackrel{\text{def}}{=} (c(Q_1, b_1), \ldots, c(Q_l, b_l), \mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$. By Property 6(b)i, there exist $store_7^o$ and l_{tok} such that

$$\begin{aligned}&\langle env^o, pe^o, stack^o, store_7^o \rangle \\ &\qquad \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s), \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \\ &store_7^o \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}}) \subseteq store^o\,.\end{aligned}$$

Let us denote $Th'' \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th^s)$ and $Th_1'' \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th_1^s)$. The thread Th_1'' is the thread Th'' where the current expression is replaced with pe_1^s. Let $Th_2^s \stackrel{\text{def}}{=} \langle env^s, (), stack^s, store^s \rangle$ be a thread intermediate between Th^s and Th_1^s, in which the result of the call has not been inserted yet in the thread. When $b_i = \text{false}$, $O'_i[\widetilde{a}_i]$ is in $\mathsf{willbeavailable}(\mathcal{CS})$, so by Property 13, $O'_i[\widetilde{a}_i]$ is not in $\mathcal{O}_{\mathsf{call}}(Th^s)$, so we can define $l'_{\mathsf{tok}} \stackrel{\text{def}}{=} l_{\mathsf{tok}|\mathcal{O}_{\mathsf{call}}(Th_2^s)} \cup \{O'_i[\widetilde{a}_i] \mapsto l_i \mid b_i = \text{false}\}$ and l'_{tok} is an extension of $l_{\mathsf{tok}|\mathcal{O}_{\mathsf{call}}(Th_2^s)}$. By Lemma B.3, Items 1 and 2, for all $\mathsf{call}(R)$ that occur in $Th_2'' \stackrel{\text{def}}{=} \mathsf{replaceinitpm}(Th_2^s)$, we have $\mathsf{correctclosure}(R, \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$. Moreover, $c(Q_i, b_i) \in$

$\mathsf{correctclosure}(O'_i[\widetilde{a}_i], \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$ for $i \leq l$, and pe^s_1 is a value so $\mathsf{replacecalls}$ allows any environment in the threads it returns, so $\langle env^o_4, pe^o_6, stack^o, store^o_7\rangle \in \mathsf{replacecalls}(Th''_1, \mathcal{I}', E_6, \mathcal{Q}_2, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$. We have

$$\begin{aligned}
&store^o_7 \cup \mathsf{gettokens}(\mathcal{I}', \mathcal{O}_{\mathsf{call}}(Th^s_1), l'_{\mathsf{tok}})\\
&\quad \subseteq store^o_7 \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}})[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \text{false}]\\
&\qquad \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\}\\
&\quad \subseteq store^o[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \text{false}] \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\}\\
&\quad \subseteq store^o_1[l_{\mathsf{tok}}(O_j[\widetilde{a}]) \mapsto \text{false}] \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\}\\
&\quad \subseteq store^o_2 \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\}\\
&\quad \subseteq store^o_5 \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} = store^o_6\,,
\end{aligned}$$

so Property 6(b)i holds with the function l'_{tok}. Properties 6(b)ii, 6(b)iii, 6(b)iv are preserved, so Property 6 holds.

Properties 7, 8, and 11 are also preserved. Properties 9, 10, 12 hold because they are kept inside Lemma 8.15. The oracles coming from $\mathsf{oracles}'(Q')$ are removed from $\mathsf{willbeavailable}(\mathcal{CS})$ and added to $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}^\infty(\mathcal{I})$; it is also removed from $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ if there remains no occurrence of $\mathsf{call}(O_j[\widetilde{a}])$ in the thread Th^s_1. So Property 13 is preserved. The oracles coming from $\mathsf{oracles}'(Q')$ are added to $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ and to $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th^s_1)$ or $\mathcal{O}_{\mathsf{call}}(Th^s_1)$ depending on whether they are under replication or not. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}_{\mathsf{call}}(Th^s_1)$ if and only if it is removed from $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$. So Property 14 is preserved. We have $|\mathcal{CT}''| + N_{\mathsf{steps}} \geq N_{\mathsf{steps}}$, so Property 15 holds. The number of calls to O_j increases by 1 and α increases by 1, so Property 16 is preserved. For the oracles $O'_i[\widetilde{a}_i]$ $(i \leq l)$, when O'_i is under replication, $O'_i\big[[1, +\infty[, \widetilde{a}\big]$ is added to $\mathcal{I}$; Property 17 is obviously satisfied for these oracles because the number of calls to these oracles is not negative. Property 17 for the previously present oracles and Property 18 are preserved, because all components of these inequalities are unchanged. So $\mathfrak{C}^{\mathsf{cs}}_1 \equiv \mathcal{CT}''$ in this case.

- If $P_4 = \mathsf{return}(M_1, \ldots, M_{m'_j})\}; Q'$, the CryptoVerif process reduces in exactly the same manner as above. The configuration $\mathcal{CS}'$ is the configuration $\mathcal{CS}$ in which we replace the current expression pe^s with $(\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j}))$, the set $\mathcal{I}$ with $\mathcal{I}' \stackrel{\text{def}}{=} \mathcal{I} \setminus \{O_j[\widetilde{a}]\}$, and the set $\mathcal{RI}$ with $\mathcal{RI}' \stackrel{\text{def}}{=} \mathcal{RI} \cup \{\mathsf{role}[\widetilde{a}] \mid (\mu_{\mathsf{role}}, \text{false}) \in \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q')\} \cup \{\mathsf{role}\big[[1, +\infty[, \widetilde{a}\big] \mid (\mu_{\mathsf{role}}, \text{true}) \in \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q')\}$.

  Let Th^o_4 be the current thread of $\mathcal{C}_4$. We have

$$\begin{aligned}
Th^o_4 = \langle &env^o_4, \mathsf{return}(\mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q'), (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{m'_j}))),\\
&stack^o, store^o_4\rangle\\
\to \langle &env^o_4, (\mathbb{G}_{\mathsf{M}}(M_1), \ldots, \mathbb{G}_{\mathsf{M}}(M_{m'_j})), stack^o_1, store^o_4\rangle\\
&\text{where } stack^o_1 \stackrel{\text{def}}{=} (env^o_4, \mathsf{return}(\mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q'), [\cdot])) :: stack^o
\end{aligned}$$

$$\rightarrow^* Th_5^o \stackrel{\text{def}}{=} \langle env_4^o, (\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), stack_1^o, store_5^o \rangle$$

by Lemma 8.13 applied m'_j times

where $store_5^o \supseteq store_4^o$ and

$\mathcal{C}_4 \rightarrow^* \mathcal{C}_4[\mathsf{Th} \mapsto Th_5^o] \rightarrow^* \mathcal{C}_5 \stackrel{\text{def}}{=} \mathcal{C}_4[\mathsf{Th} \mapsto Th_6^o, \mathsf{MI} \mapsto \mathcal{MI}']$ where $Th_6^o \stackrel{\text{def}}{=} \langle env_4^o, (\mathbb{G}_{\mathsf{val}T'_{j,1}}(c_1), \ldots, \mathbb{G}_{\mathsf{val}T'_{j,m'_j}}(c_{m'_j})), stack^o, store_5^o \rangle$ and $\mathcal{MI}' \stackrel{\text{def}}{=} \mathcal{MI} \cup \mathbb{G}_{\mathsf{get}\mathcal{MI}}(Q')$.

Let $\mathcal{CT}''$ be the trace $\mathcal{CT}$ followed by $\mathcal{CT}'$ and extended until $\mathcal{C}_5$.

Let us now prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$. By the form of $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{C}''$, Properties 1 and 2 hold. The set $\mathcal{Q}_2$ is the set $\mathcal{Q}$ from which we removed the oracles $O_j[\widetilde{a}]$ and $O_{\text{loop}}[\alpha+1]$ and to which we added the new oracles of reduce$'(Q')$. The set $\mathcal{I}'$ is the set $\mathcal{I}$ from which we removed $O_j[\widetilde{a}]$. We added the elements of $\mathsf{oracles}'(Q')$ to $\mathcal{O}(\mathcal{RI}')$. So Property 3 is preserved. Properties 4 to 10, 12, and 15 to 18 are proved as in the case $P_4 = \mathsf{end}$. We added matching elements in $\mathcal{MI}'$ and in $\mathcal{RI}'$, so Property 11 is preserved. The oracles coming from $\mathsf{oracles}'(Q')$ are removed from $\mathsf{willbeavailable}(\mathcal{CS})$ and added to $\mathcal{O}^\infty(\mathcal{RI})$. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}^\infty(\mathcal{I})$; it is also removed from $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ if there remains no occurrence of $\mathsf{call}(O_j[\widetilde{a}])$ in the thread Th_1^s. So Property 13 is preserved. The oracle $O_j[\widetilde{a}]$ is removed from $\mathcal{O}_{\mathsf{call}}(Th_1^s)$ if and only if it is removed from $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$, and the other oracle sets of Property 14 are unchanged, so Property 14 is preserved. So $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}''$ in this case.

Case 2.2. The current expression of $\mathcal{CS}$ is $pe^s = \mathsf{call}(O_j[\_, \widetilde{a}])\ (v_1, \ldots, v_{m_j})$ and $\mathcal{CS}$ cannot reduce, that is, the configuration $\mathcal{CS}$ makes a successful call to $O_j[\_, \widetilde{a}]$, an oracle under replication. We prove this case by a reasoning similar to the previous case.

We show that a copy of the oracle $O_j[\_, \widetilde{a}]$ is available in $\mathcal{Q}$ using Property 17, as follows. By Property 6(b)i, $pe^o = \mathsf{tagfunction}^{O_j,\tau}[env_1^o, pm_{\text{true}}(Q)]\ (v_1, \ldots, v_{m_j})$, with $\tau = \tau_{\mathsf{O}}(O_j[\_, \widetilde{a}])$ and $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$ for some a'. Let $\mathcal{CT}'$ be the extension of $\mathcal{CT}$ with one step. By definition of N_{O_j}, we have $N_{O_j} \geq N_{\mathsf{calls}}(O_j, \tau, \mathcal{CT}') = N_{\mathsf{calls}}(O_j, \tau, \mathcal{CT}) + 1$, so by Property 17, $a' \leq N_{O_j}$, so $O_j[a', \widetilde{a}] \in \mathcal{O}(\mathcal{I})$, so by Property 3, $\mathcal{Q}_0$ contains a process of the form $O_j[a', \widetilde{a}](x_1[a', \widetilde{a}] : T_{j,1}, \ldots, x_{m_j}[a', \widetilde{a}] : T_{j,m_j}) := P_O$.

Due to the call, the index a' such that $O\big[[a', +\infty[, \widetilde{a}\big] \in \mathcal{I}$ increases by 1 and the number of calls to the closure with tag O_j, τ increases by 1, so Property 17 is preserved.

Case 2.3. The current expression of $\mathcal{CS}$ is $pe^s = \mathsf{random}\ ()$, that is, the configuration $\mathcal{CS}$ samples a random boolean. By Property 6, the current expression of $\mathcal{C}$ is $pe^o = \mathsf{random}\ ()$. For $b \in \{\text{true}, \text{false}\}$, $\mathcal{C} \rightarrow_{1/2} \mathcal{C}_b$ where $\mathcal{C}_b \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env^o, b, stack^o, store^o \rangle]$. Let $\mathcal{CT}_b$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_b$.

The configuration $\mathcal{CS}$ cannot reduce, and $simreturn(\mathcal{CS}) = (\mathsf{repr}(\mathcal{CS}), o_R, (), ())$. Let us denote $s \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{CS})$. The simulator configuration reduces in the

following way for a CryptoVerif value $b \in \{\text{true}, \text{false}\}$.

$$\begin{aligned}
\mathfrak{C}^{\mathsf{cs}} &\rightsquigarrow^* E_1, P'_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\quad \text{where } E_1 \stackrel{\text{def}}{=} E[xs[\alpha] \mapsto (s, o_R, (), ()), s'[\alpha] \mapsto s, o[\alpha] \mapsto o_R, \\
&\qquad\qquad i[\alpha] \mapsto (), args[\alpha] \mapsto ()] \\
&\rightsquigarrow^* E_1, P^R_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\rightsquigarrow_{1/2} E_2, \mathsf{return}(simulate_R(s'[\alpha], b_R[\alpha]), \text{true}), \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\quad \text{where } E_2 \stackrel{\text{def}}{=} E_1[b_R[\alpha] \mapsto b] \\
\rightsquigarrow^* \mathfrak{C}^{\mathsf{cs}}_b &\stackrel{\text{def}}{=} E_3, P_{loop}\{\alpha+1/i'\}, \mathcal{T}, \mathcal{Q}_1, \mathcal{R}_{loop}(\alpha+1), \mathcal{E}, N_{\mathsf{steps}}, \mathcal{CS}_b \\
&\quad \text{where } E_3 \supseteq E_2[s[\alpha+1] \mapsto \mathsf{repr}(\mathcal{CS}_b)], \\
&\qquad\qquad \mathcal{Q}_1 \stackrel{\text{def}}{=} \mathcal{Q} \setminus \{Q_{loop}\{\alpha+1/i'\}\}, \\
&\qquad\qquad \mathcal{CS}_b \stackrel{\text{def}}{=} \mathcal{CS}[\mathsf{Th} \mapsto \langle env^s, \mathbb{G}_{\mathsf{val}bool}(b), stack^s, store^s\rangle]
\end{aligned}$$

We verify that $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}$ using Property 16, as follows. By definition of $N_{\mathsf{rand+calls}}$, $N_{\mathsf{rand+calls}} \geq \left(N_{\mathsf{rand}}(\mathcal{CT}_b) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT}_b)\right) + 1 = \left(N_{\mathsf{rand}}(\mathcal{CT}) + \sum_{O,\tau} N_{\mathsf{calls}}(O, \tau, \mathcal{CT})\right) + 2$ since $\mathcal{CT}_b$ makes one more random number generation than $\mathcal{CT}$. So by Property 16, $N_{\mathsf{rand+calls}} \geq \alpha + 1$. So by Property 3, $Q_{loop}\{\alpha+1/i'\} \in \mathcal{Q}$. Moreover, we have $\mathbb{G}_{\mathsf{val}bool}(b) = b$, so $\mathcal{CS}_b = \mathcal{CS}[\mathsf{Th} \mapsto \langle env^s, b, stack^s, store^s\rangle]$.

In this step, α becomes $\alpha + 1$, the number of random number generations in the trace increases by 1, the current thread is modified exactly in the same manner on both sides, and the other threads, the oracle sets, the global store, and the events are left unchanged, so it is easy to see that $\mathfrak{C}^{\mathsf{cs}}_{\text{true}} \equiv \mathcal{CT}_{\text{true}}$ and $\mathfrak{C}^{\mathsf{cs}}_{\text{false}} \equiv \mathcal{CT}_{\text{false}}$.

Case 2.4. The configuration $\mathcal{CS}$ does not reduce, and does not make a call to an oracle nor sample a random boolean. In this case, $simreturn(\mathcal{CS}) = (\mathsf{repr}(\mathcal{CS}), o_S, (), ())$. Let us denote $s \stackrel{\text{def}}{=} \mathsf{repr}(\mathcal{CS})$. The simulator configuration reduces in the following way.

$$\begin{aligned}
\mathfrak{C}^{\mathsf{cs}} &\rightsquigarrow^* E_1, P'_{loop}\{\alpha/i'\}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\quad \text{where } E_1 \stackrel{\text{def}}{=} E[xs[\alpha] \mapsto (s, o_S, (), ()), s'[\alpha] \mapsto s, o[\alpha] \mapsto o_S, i[\alpha] \mapsto (), \\
&\qquad\qquad args[\alpha] \mapsto ()] \\
&\rightsquigarrow E_1, \mathsf{return}(s'[\alpha], \text{false}), \mathcal{T}, \mathcal{Q}, \mathcal{R}_{loop}(\alpha), \mathcal{E} \\
&\rightsquigarrow E_2, P_{return\text{-}loop}(\alpha), \mathcal{T}, \mathcal{Q}, \mathcal{R}_1, \mathcal{E} \\
&\quad \text{where } E_2 \stackrel{\text{def}}{=} E_1[r'_{\alpha,r}[\,] \mapsto s, b_{\alpha,r}[\,] \mapsto \text{false}], \\
&\qquad\qquad \mathcal{R}_1 \stackrel{\text{def}}{=} [x[\,], \mathsf{return}(x[\,]), \mathsf{end}] \\
&\rightsquigarrow E_2, r[\,] \leftarrow r'_{\alpha,r}[\,]; \mathsf{end}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_1, \mathcal{E} \\
&\rightsquigarrow E_3, \mathsf{end}, \mathcal{T}, \mathcal{Q}, \mathcal{R}_1, \mathcal{E} \\
&\quad \text{where } E_3 \stackrel{\text{def}}{=} E_2[r[\,] \mapsto s] \\
\rightsquigarrow \mathfrak{C}^{\mathsf{cs}}_1 &\stackrel{\text{def}}{=} E_3, \mathsf{end}, \mathcal{T}, \mathcal{Q}, [\,], \mathcal{E}
\end{aligned}$$

This configuration cannot reduce. By Property 6, the OCaml configuration $\mathcal{C}$ also cannot reduce. (If it could reduce, then the simulator configuration $\mathcal{CS}$ would reduce by the same rule as the OCaml configuration.) Moreover, by Property 12 of the invariant, $events = \mathbb{G}_{\mathsf{ev}}(\mathcal{E})$, so this case satisfies the second point of Lemma 8.34.

Case 2.5. The current expression of $\mathcal{CS}$ is of the form $pe^s = \mathsf{tagfunction}^{t,\tau}[env, pm]\ v$, that is, $\mathcal{CS}$ calls a tagged closure. By Property 6(b)iii, the only tagged closures present in the current thread are of the form $\mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^s, pm'_{\mathsf{role}[\widetilde{a}]}]$ for a given role $\mathsf{role}[\widetilde{a}]$, with $env_{\mathsf{prim}} \subseteq env_1^s$. Such a closure corresponds to the initialization of the role $\mathsf{role}[\widetilde{a}]$. Since our programs are well-typed, and these closures expect an argument of type unit, the current expression of $\mathcal{CS}$ is $pe^s = \mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^s, pm'_{\mathsf{role}[\widetilde{a}]}]\ ()$.

Let us denote by Q_i, b_i for $i \leq m$ the oracles present in $\mathsf{reduce}'(Q(\mathsf{role})[\widetilde{a}])$, and let $\widetilde{a_i} = \widetilde{a}$ or $\_, \widetilde{a}$ such that $O_i'[\widetilde{a_i}]$ is the oracle associated to Q_i, b_i in $\mathsf{oracles}'(Q(\mathsf{role})[\widetilde{a}])$.

By Property 6(b)i,

$$pe^o = \mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^o, pm_{\mu_{\mathsf{role}}}]\ ()\,.$$

By Property 6(b)ii, $env_1^s(token) = l_{\mathsf{init\text{-}tok}}(\mathsf{role}[\widetilde{a}])$ is a location. Let us denote l this location. By Property 6(b)i, we have $env_1^o(token) = env_1^s(token) = l$. By Property 5, l is in the domain of $store^s$. By Property 6(b)i, l is also in the domain of $store^o$.

Let $x_1[\,], \ldots, x_k[\,]$ be the free variables of the role role.

Let us denote

$$\begin{aligned} pe_e^s &\stackrel{\text{def}}{=} (\mathsf{call}(O_1'[\widetilde{a_1}]), \ldots, \mathsf{call}(O_m'[\widetilde{a_m}])) \\ pe_e^o &\stackrel{\text{def}}{=} \mathbb{G}_{\mathsf{read}}(x_1[\,])\ \mathsf{in}\ \ldots\ \mathsf{in}\ \mathbb{G}_{\mathsf{read}}(x_k[\,])\ \mathsf{in} \\ &\qquad (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_m, b_m)) \end{aligned}$$

The simulator reduces as follows:

$$\begin{aligned} Th^s &= \langle env^s, \mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^s, pm'_{\mathsf{role}}]\ (), stack^s, store^s\rangle \\ &\to \langle env_1^s, \mathsf{match}\ ()\ \mathsf{with}\ pm'_{\mathsf{role}}, stack^s, store^s\rangle \\ \to Th_1^s &\stackrel{\text{def}}{=} \langle env_1^s, pe_1^s, stack^s, store^s\rangle \\ &\qquad \text{where } pe_1^s \stackrel{\text{def}}{=} \mathsf{if}\ !token\ \mathsf{then}\ (token := \mathrm{false}; pe_e^s)\ \mathsf{else\ raise\ Bad\_Call} \end{aligned}$$

and the OCaml side reduces as follows:

$$\begin{aligned} Th^o &= \langle env^o, \mathsf{tagfunction}^{\mathsf{role},\tau}[env_1^o, pm_{\mu_{\mathsf{role}}}]\ (), stack^o, store^o\rangle \\ &\to \langle env_1^o, \mathsf{match}\ ()\ \mathsf{with}\ pm_{\mu_{\mathsf{role}}}, stack^o, store^o\rangle \\ \to Th_1^o &\stackrel{\text{def}}{=} \langle env_1^o, pe_1^o, stack^o, store^o\rangle \\ &\qquad \text{where } pe_1^o \stackrel{\text{def}}{=} \mathsf{if}\ !token\ \mathsf{then}\ (token := \mathrm{false}; pe_e^o)\ \mathsf{else\ raise\ Bad\_Call} \end{aligned}$$

- If $store^s(l) = \text{false}$, then by Property 6(b)i, $store^o(l) = \text{false}$, so

$$Th_1^s \to^* Th_2^s \stackrel{\text{def}}{=} \langle env_1^s, \mathsf{raise\ Bad\_Call}, stack^s, store^s \rangle$$
$$Th_1^o \to^* Th_2^o \stackrel{\text{def}}{=} \langle env_1^o, \mathsf{raise\ Bad\_Call}, stack^s, store^s \rangle$$

  Let $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}[\mathsf{Th} \mapsto Th_2^o]$, $\mathcal{CS}_1 \stackrel{\text{def}}{=} \mathcal{CS}[\mathsf{Th} \mapsto Th_2^s]$, $steps^s$ the number of steps of the trace $\mathcal{CS} \to^* \mathcal{CS}_1$, and $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - steps^s, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathfrak{C}_1^{\mathsf{cs}}$.

  Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. As the current expression is an exceptional value, replacecalls allows any environment in its image. Moreover, the other elements of the configuration are the same and $\mathcal{I}$ did not change, so Property 6 is preserved. The number of steps in the reduction is the same on both sides, so Property 15 is preserved. All other properties of Definition 8.32 are trivially inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

- Otherwise, $store^s(l) = \text{true}$. By Property 6(b)i, $store^o(l) = \text{true}$.

  On the simulator side:

$$Th_1^s \to^* Th_3^s \stackrel{\text{def}}{=} \langle env_1^s, pe_e^s, stack^s, store_1^s \rangle$$
$$\text{where } store_1^s \stackrel{\text{def}}{=} store^s[l \mapsto \text{false}]$$

  By Property 4, the variables $x_1[\,], \ldots, x_k[\,]$ are present in the environment E. Let $a'_1, \ldots, a'_k$ be the values of these variables in the environment E. By Property 9, $globalstore(E, \mathcal{T}) \subseteq globalstore^o$, so $globalstore^o(f_i) = \mathsf{ser}(T_{x_i}, a'_i)$ where $(x_i[\,], f_i) \in \mathsf{files}$ for all $i \leq k$. Let $\mathcal{C}_1 \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto Th_1^o]$. We have

$$\begin{aligned}
\mathcal{C}_1 &\to^* \mathcal{C}[\mathsf{Th} \mapsto \langle env_1^o, pe_e^o, stack^o, store_1^o \rangle] \\
&\quad \text{where } store_1^o \stackrel{\text{def}}{=} store^o[l \mapsto \text{false}] \\
&\to^* \mathcal{C}_2 \stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env_2^o, pe_2^o, stack^o, store_2^o \rangle] \\
&\quad \text{where } pe_2^o \stackrel{\text{def}}{=} (\mathbb{G}_{\mathsf{O}}(Q_1, b_1), \ldots, \mathbb{G}_{\mathsf{O}}(Q_m, b_m)), \\
&\qquad env_2^o \stackrel{\text{def}}{=} env_1^o[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto \mathbb{G}_{\mathsf{val}T_{x_1}}(a'_1), \ldots, \\
&\qquad\qquad \mathbb{G}_{\mathsf{var}}(x_k) \mapsto \mathbb{G}_{\mathsf{val}T_{x_k}}(a'_k)], \\
&\qquad store_2^o \supseteq store_1^o
\end{aligned}$$

  by Proposition 8.5 applied k times to show the correctness of the deserialization primitives.

  Let $l_1, \ldots, l_m$ be pairwise distinct locations that are not in $\text{Dom}(store_2^o)$ and $\tau_1, \ldots, \tau_m$ be pairwise distinct fresh tags. By the same reasoning as in Case 2.1, sub-case $P_4 = \mathsf{return}(M_1, \ldots, M_{m'_j}); Q'$, we have

$$\begin{aligned}
\mathcal{C}_2 \to^* \mathcal{C}_3 &\stackrel{\text{def}}{=} \mathcal{C}[\mathsf{Th} \mapsto \langle env_2^o, pe_3^o, stack^o, store_3^o \rangle] \\
&\quad \text{where } pe_3^o \stackrel{\text{def}}{=} (\mathsf{tagfunction}^{O'_1, \tau_1}[env_{c,1}^o, pm_{b_1}(Q_1)], \ldots, \\
&\qquad\qquad \mathsf{tagfunction}^{O'_m, \tau_m}[env_{c,m}^o, pm_{b_m}(Q_m)]), \\
&\qquad store_3^o \stackrel{\text{def}}{=} store_2^o \cup \{l_i \mapsto \text{true} \mid i \leq m, b_i = \text{false}\}
\end{aligned}$$

where, for all $i \leq m$, $env^o_{c,i}$ is env^o_2 when b_i is true and $env^o_2[token \mapsto l_i]$ otherwise.

Let $\mathcal{CT}_2$ be an extension of the trace $\mathcal{CT}$ until $\mathcal{C}_3$. Let $\mathcal{CS}_3 \stackrel{\text{def}}{=} \mathcal{CS}[\mathsf{Th} \mapsto Th^s_3]$. Let $steps^s$ be the number of steps of $\mathcal{CS} \to^* \mathcal{CS}_3$. Let $\mathfrak{C}^{\mathsf{cs}}_2 \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - steps^s, \mathcal{CS}_3$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow^+ \mathfrak{C}^{\mathsf{cs}}_2$.

Let us prove that $\mathfrak{C}^{\mathsf{cs}}_2 \equiv \mathcal{CT}_2$. We define τ'_{O} as τ_{O} except that for all $i \leq m$, if $b_i = \text{true}$, then $\tau'_{\mathsf{O}}(O'_i[\_, \widetilde{a}]) = \tau_i$. Properties 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 16, 18 hold because they hold for $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

Let us prove Property 6. First, we prove Property 6(b) for the current thread. For all $i \leq m$, the free variables of Q_i are contained in $\{x_1[\,], \ldots, x_k[\,]\}$, so $env^o_{c,i} \supseteq env(E, Q_i)$. Moreover, by Properties 6(b)iii and 6(b)i, $env_{\mathsf{prim}} \subseteq env^o_1$, so $env^o_{c,i} \supseteq env^o_2 \supseteq env^o_1 \supseteq env_{\mathsf{prim}}$. We have $\mathsf{role}[\widetilde{a}] \in \mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s)$. By Property 14, $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$ is included in $\mathcal{O}^\infty(\mathcal{I}) \cup \mathcal{O}_{\mathsf{call}}(\mathcal{CS})$, and furthermore $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$ is disjoint from $\mathcal{O}_{\mathsf{call}}(Th_i)$ for all $i \leq n$, so from $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$, so $\mathcal{O}^\infty(\{\mathsf{role}[\widetilde{a}]\})$ is included in $\mathcal{O}^\infty(\mathcal{I})$. Hence, when O'_i is not under replication (that is, $b_i = \text{false}$), $O'_i[\widetilde{a_i}] \in \mathcal{I}$, and when O'_i is under replication, $\widetilde{a_i} = \_, \widetilde{a}$ and $O'_i\big[[1, +\infty[, \widetilde{a}\big] \in \mathcal{I}$. By Property 3, when O'_i is not under replication, $Q_i = \mathcal{Q}(O'_i[\widetilde{a_i}])$, and when O'_i is under replication, $Q_i = \mathcal{Q}(O'_i[1, \widetilde{a}])$.

By Property 6(b)i, there exist $store^o_4$ and l_{tok} such that

$$\begin{aligned}
&\langle env^o, pe^o, stack^o, store^o_4\rangle \\
&\qquad \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s), \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) \\
&store^o_4 \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}}) \subseteq store^o\,.
\end{aligned}$$

Since $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$ is disjoint from $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$ as noticed above, the oracles $O'_i[\widetilde{a_i}]$ are not present in $\mathcal{O}_{\mathsf{call}}(\mathcal{CS})$. So we can define the injective function $l'_{\mathsf{tok}} \stackrel{\text{def}}{=} l_{\mathsf{tok}} \cup \{O'_i[\widetilde{a_i}] \mapsto l_i \mid i \leq m, b_i = \text{false}\}$. By Lemma B.3, Item 2, for all $\mathsf{call}(R)$ that occur in $\mathsf{replaceinitpm}(Th^s)$, $\mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}}) \supseteq \mathsf{correctclosure}(R, \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$, noticing that, when $i \leq m$ and $b_i = \text{true}$, $O'_i[N_{O'_i} + 1, \widetilde{a}] \in \mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$, so by Property 14, $O'_i[N_{O'_i} + 1, \widetilde{a}] \notin \mathcal{O}_{\mathsf{call\text{-}repl}}(Th^s)$, so $\mathsf{call}(O'_i[\_, \widetilde{a}])$ does not occur in $\mathsf{replaceinitpm}(Th^s)$, so the transformation of τ_{O} into τ'_{O} does not affect the computation of these correct closures. Moreover, $\mathsf{tagfunction}^{O'_i, \tau_i}[env^o_{c,i}, pm_{b_i}(Q_i)] \in \mathsf{correctclosure}(O'_i[\widetilde{a_i}], \mathcal{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$ for $i \leq m$ and pe^s_e is a value so $\mathsf{replacecalls}$ allows any environment in the threads it returns, so $\langle env^o_2, pe^o_3, stack^o, store^o_4[l \mapsto \text{false}]\rangle \in \mathsf{replacecalls}(\mathsf{replaceinitpm}(Th^s_3), \mathcal{I}, E, \mathcal{Q}, l'_{\mathsf{tok}}, \tau'_{\mathsf{O}})$. We have

$$\begin{aligned}
&store^o_4[l \mapsto \text{false}] \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s_3), l'_{\mathsf{tok}}) \\
&\quad \subseteq store^o_4[l \mapsto \text{false}] \cup \mathsf{gettokens}(\mathcal{I}, \mathcal{O}_{\mathsf{call}}(Th^s), l_{\mathsf{tok}}) \\
&\qquad \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} \\
&\quad \subseteq store^o[l \mapsto \text{false}] \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} \\
&\quad \subseteq store^o_1 \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} \\
&\quad \subseteq store^o_2 \cup \{l_i \mapsto \text{true} \mid b_i = \text{false}\} = store^o_3\,,
\end{aligned}$$

so Property 6(b)i holds with the function l'_{tok}. Properties 6(b)ii, 6(b)iii, 6(b)iv are preserved, so Property 6 holds for the current thread. The other threads and $\mathcal{I}, E, \mathcal{Q}$ are unchanged, and as above, the transformation of τ_{O} into τ'_{O} does not affect the computation of correct closures in these threads, so Property 6 holds for all threads.

The role $\mathsf{role}[\widetilde{a}]$ is removed from $\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s)$, so the elements added to $\mathcal{O}_{\mathsf{call}}(Th^s)$ and $\mathcal{O}_{\mathsf{call\text{-}repl}}(Th^s)$ are removed from $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}closure}}(Th^s))$, hence Properties 13 and 14 are preserved. There are more steps on the OCaml side than on the CryptoVerif side, so Property 15 is preserved. For the oracles $O'_i[\widetilde{a_i}]$ $(i \leq l)$, when O'_i is under replication, we have already shown that $O'_i\big[[1, +\infty[, \widetilde{a}\big] \in \mathcal{I}$; Property 17 is obviously satisfied for these oracles because the number of calls to these oracles is not negative. Property 17 is preserved for the other oracles, because all components of these inequalities are unchanged.

Case 2.6. The current expression of $\mathcal{CS}$ is $pe^s = \mathsf{addthread}(program)$, that is, we add a new thread to the current configuration. By Property 6(b)i, the expression pe^o is $\mathsf{addthread}(program)$, and by Property 6(b)iv, *program* contains no closure, no tagged function, no event, no return except in parts $program(\mu_{\mathsf{role}})$, and in $program(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$.

Suppose first that *program* is an attacker program: it does not contain $program(\mu_{\mathsf{role}})$ except in arguments of $\mathsf{addthread}$. In this case,

$$\begin{aligned}\mathcal{CS} \to \mathcal{CS}_1 \stackrel{\text{def}}{=} ([Th_1, \ldots, Th_{tj-1}, \langle env^s, (), stack^s, store^s\rangle, Th_{tj+1}, \ldots, Th_n,\\ \langle \emptyset, program, [\,], \emptyset\rangle], globalstore^s, tj), \mathcal{RI}, \mathcal{I}\\ \mathcal{C} \to \mathcal{C}_1 \stackrel{\text{def}}{=} [Th'_1, \ldots, Th'_{tj-1}, \langle env^o, (), stack^o, store^o\rangle, Th_{tj+1}, \ldots, Th_n,\\ \langle \emptyset, program, [\,], \emptyset\rangle], globalstore^o, tj, \mathcal{MI}, events\end{aligned}$$

Let $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_1$ and $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - 1, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}}$. Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. The new thread contains no closures and no tagged functions. It contains no call since *program* is an OCaml program (not a simulator program), so it satisfies Property 6(b). The other properties are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

Otherwise, the program *program* is of the form

$$program_{\mathsf{prim}};; program(\mu_{\mathsf{role}_1});; \ldots;; program(\mu_{\mathsf{role}_m});; program'\,,$$

where *program*′ does not contain $program(\mu)$ for any $\mu \in \mathcal{M}_g$. By Assumption 6.1, for $\mathcal{M} \stackrel{\text{def}}{=} \{\mu_{\mathsf{role}_1}, \ldots, \mu_{\mathsf{role}_m}\}$, we have $\mathcal{M} \subseteq \mathcal{M}_g$ and $\forall \mu \in \mathcal{M}, \exists b$, $(\mu, b) \in \mathcal{MI}$. By Property 11, for each $i \leq m$, if role_i is not under replication, then the set $\mathcal{RI}$ contains $\mathsf{role}_i[\widetilde{a}]$ for some $\widetilde{a}$, and if role_i is under replication, then the set $\mathcal{RI}$ contains $\mathsf{role}_i\big[[a', +\infty[, \widetilde{a}\big]$ for some $a', \widetilde{a}$. By Property 13, the

oracles present in $\mathcal{RI}$ are not in $\mathcal{I}$. We can then define

$$\widetilde{a_1} \stackrel{\text{def}}{=} \mathsf{smallest}(\mathcal{RI}, \mathsf{role}_1), \ldots, \widetilde{a_m} \stackrel{\text{def}}{=} \mathsf{smallest}(\mathcal{RI}, \mathsf{role}_m)$$
$$\mathcal{RI}'' \stackrel{\text{def}}{=} \{\mathsf{role}_1[\widetilde{a_1}], \ldots, \mathsf{role}_m[\widetilde{a_m}]\}$$
$$\mathcal{RI}' \stackrel{\text{def}}{=} \mathcal{RI} - \mathcal{RI}'' \qquad \mathcal{I}' \stackrel{\text{def}}{=} \mathsf{add}(\mathcal{I}, \mathcal{RI}'')$$
$$\mathit{program}^b \stackrel{\text{def}}{=} \mathit{program}_{\mathsf{prim}};; \mathit{program}'(\mathsf{role}_1[\widetilde{a_1}]);; \ldots;; \mathit{program}'(\mathsf{role}_m[\widetilde{a_m}]);; \mathit{program}'$$
$$\mathcal{MI}' \stackrel{\text{def}}{=} \{(\mu, \text{false}) \mid \mu \in \mathcal{M} \wedge (\mu, \text{false}) \in \mathcal{MI}\}$$

We have

$$\mathcal{CS} \to \mathcal{CS}_2 \stackrel{\text{def}}{=} ([\mathit{Th}_1, \ldots, \mathit{Th}_{tj-1}, \langle \mathit{env}^s, (), \mathit{stack}^s, \mathit{store}^s\rangle, \mathit{Th}_{tj+1}, \ldots, \mathit{Th}_n, \langle \emptyset, \mathit{program}^b, [\,], \emptyset\rangle], \mathit{globalstore}^s, tj), \mathcal{RI}', \mathcal{I}'$$
$$\mathcal{C} \to \mathcal{C}_2 \stackrel{\text{def}}{=} [\mathit{Th}'_1, \ldots, \mathit{Th}'_{tj-1}, \langle \mathit{env}^o, (), \mathit{stack}^o, \mathit{store}^o\rangle, \mathit{Th}_{tj+1}, \ldots, \mathit{Th}_n, \langle \emptyset, \mathit{program}, [\,], \emptyset\rangle], \mathit{globalstore}^o, tj, \mathcal{MI} \setminus \mathcal{MI}', \mathit{events}$$

Let $\mathcal{CT}_2$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_2$ and $\mathfrak{C}_2^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, \mathit{steps} - 1, \mathcal{CS}_2$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_2^{\mathsf{cs}}$. Let us prove that $\mathfrak{C}_2^{\mathsf{cs}} \equiv \mathcal{CT}_2$. The oracles under replication added to $\mathcal{I}$ are the oracles $O\big[[1, +\infty[, \widetilde{a_i}\big]$ such that $O[\_, \widetilde{a_i}] \in \mathsf{oracles}'(Q(\mathsf{role}_i)[\widetilde{a_i}])$ for any $i \leq m$. We define τ'_{O} as the extension of τ_{O} that maps all the oracles $O[\_, \widetilde{a_i}]$ to fresh distinct tags τ. Properties 1, 2, 4, 8, 9, 10, 12, 15, and 16 are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$. By Property 3, $\mathcal{Q} = \{Q_{\mathit{loop}}\{a/i'\} \mid \alpha < a \leq N_{\mathsf{rand}+\mathsf{calls}}\} \cup \mathcal{Q}_0$ and $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}, \mathcal{I}$. If role_i is under replication, then by definition of $\mathsf{smallest}$, $\mathsf{role}_i\big[[a'_i, +\infty[, \widetilde{a''_i}\big] \in \mathcal{RI}$ with $\widetilde{a_i} = a'_i, \widetilde{a''_i}$. By definition of N_{role_i}, $N_{\mathsf{role}_i} \geq N_{\mathsf{exec}}(\mathsf{role}_i, \mathcal{CT}_2) = N_{\mathsf{exec}}(\mathsf{role}_i, \mathcal{CT})+1$. By Property 18, $a'_i \leq N_{\mathsf{role}_i}$. Therefore, the set $\mathcal{O}(\mathcal{RI})$ contains the first oracles of $\mathsf{role}_i[\widetilde{a_i}]$ for $i \leq m$. The set $\mathcal{O}(\mathcal{RI}')$ is the set $\mathcal{O}(\mathcal{RI})$ from which we remove the first oracles of $\mathsf{role}_i[\widetilde{a_i}]$ for $i \leq m$ and $\mathcal{O}(\mathcal{I}')$ is the set $\mathcal{O}(\mathcal{I})$ to which we add these oracles. So $\mathcal{Q}_0 \leftrightarrow \mathcal{RI}', \mathcal{I}'$ and Property 3 holds. There are no local store locations in *program*, so Property 5 holds. For each thread Th_i of the simulator except the new thread, let us show that Property 6 is preserved. The only changes are that the current expression is replaced with () and that $\mathcal{I}' = \mathsf{add}(\mathcal{I}, \mathcal{RI}'')$, so we just have to show that Property 6(b)i is preserved; the other elements of Property 6 are obviously preserved. By Lemma B.3, Item 2, the correct closures are preserved. By Property 14, the set $\mathcal{O}_{\mathsf{call}}(\mathit{Th}_i)$ does not contain any of the oracles added to $\mathcal{I}$, so the tokens are preserved. Hence, Property 6(b)i is preserved. Since $\mathit{program}'$ already occurs in the initial program, it does not contain closures. By Property 6(b)iv, it does not contain tagged functions, events, or returns, except in $\mathit{program}(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$, so Property 6(a) holds for the new thread. By Property 7, $\mathit{program}'$ does not contain any location $l \in S_{priv}$ except in $\mathit{program}(\mu_{\mathsf{role}})$ in arguments of $\mathsf{addthread}$, so Property 7 holds. When role_i is not under replication, we remove one copy of the module μ_{role_i} from the multiset $\mathcal{MI}$, and correspondingly, we remove $\mathsf{role}_i[\widetilde{a_i}]$ from $\mathcal{RI}$. When role_i is under replication, we add 1 to the first index of roles $\mathsf{role}_i[\widetilde{a_i}]$ in $\mathcal{RI}$, and $\mathcal{MI}$ is not affected by this change. (The

role role_i can still be called.) So Property 11 is preserved. The first oracles of $\mathsf{role}_1[\widetilde{a_1}], \ldots, \mathsf{role}_m[\widetilde{a_m}]$ are transferred from $\mathcal{O}^\infty(\mathcal{RI})$ to $\mathcal{O}^\infty(\mathcal{I})$, so Property 13 is preserved. More precisely, these oracles are added to $\mathcal{O}^\infty(\mathcal{R}_{\mathsf{init\text{-}function}}(Th_{n+1}))$, where $Th_{n+1} \stackrel{\text{def}}{=} \langle \emptyset, program^b, [\,], \emptyset \rangle$ is the new thread, so Property 14 is preserved. For the oracles $O\big[[1, +\infty[, \widetilde{a_i}\big]$ added to $\mathcal{I}$, Property 17 is obviously satisfied because the number of calls to an oracle is not negative. Property 17 is preserved for the previously present oracles, because all components of these inequalities are unchanged. For the roles $\mathsf{role}_i\big[[a'_i, +\infty[, \widetilde{a''_i}\big] \in \mathcal{RI}$, with $\widetilde{a_i} = a'_i, \widetilde{a''_i}$, we have $\mathsf{role}_i\big[[a'_i + 1, +\infty[, \widetilde{a''_i}\big] \in \mathcal{RI}$; the elements $\mathsf{role}_i[\ldots]$ in $\mathcal{RI}$ with indices that do not end with $\widetilde{a''_i}$ are unchanged; and $N_{\mathsf{exec}}(\mathsf{role}_i, \mathcal{CT})$ increases by 1, so Property 18 is preserved for the roles $\mathsf{role}_1, \ldots, \mathsf{role}_m$. Property 18 is preserved for the other roles, because all components of the inequalities are unchanged. Therefore, $\mathfrak{C}_2^{\mathsf{cs}} \equiv \mathcal{CT}_2$.

Case 2.7. The current expression of $\mathcal{CS}$ is of the form $pe^s = \mathsf{call}(O[\widetilde{a}])\ v$ and $\mathcal{CS}$ reduces, that is, we call an oracle but the call fails. By reduction rule (FailedCall1) or (FailedCall2),

$$Th^s \to Th_1^s \stackrel{\text{def}}{=} \langle env^s, \mathsf{raise\ Bad\_Call}, stack^s, store^s \rangle,$$

and $\mathcal{CS} \to \mathcal{CS}_1 \stackrel{\text{def}}{=} \mathcal{CS}[\mathsf{Th} \mapsto Th_1^s]$.

By Property 6(b)i, $pe^o = c\ v'$, where $c \in \mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}})$.

We suppose that the program is well typed, so the value v is a k-tuple $(v_1, \ldots, v_k)$, where k is the number of arguments of oracle O. Let $T_1, \ldots, T_k$ be their CryptoVerif types. Let $x_1, \ldots, x_k$ be the CryptoVerif variables that are the arguments of O. By Assumption 8.3, the value v' does not contain closures nor locations, so $v' = v$.

Let us first suppose that the oracle O is under replication. In this case, $\widetilde{a} = \_, \widetilde{a'}$. There exists a'' such that $O\big[[a'', +\infty[, \widetilde{a'}\big] \in \mathcal{I}$, because otherwise we would have $\mathsf{correctclosure}(O[\widetilde{a}], \mathcal{I}, E, \mathcal{Q}, l_{\mathsf{tok}}, \tau_{\mathsf{O}}) = \emptyset$. The closure c is of the form $\mathsf{tagfunction}^{O,\tau}[env_1^o, pm_{\text{true}}(Q)]$. Let $\mathcal{CT}'$ be the extension of $\mathcal{CT}$ by one step. By definition of N_O, $N_O \geq N_{\mathsf{calls}}(O, \tau, \mathcal{CT}') = N_{\mathsf{calls}}(O, \tau, \mathcal{CT}) + 1$. Hence, by Property (17), $a'' \leq N_O$. Therefore, by definition of $\mathsf{correctclosure}$, $Q = \mathcal{Q}(O[a'', \widetilde{a'}])$. Since (FailedCall2) applies, there exists i such that $\forall a \in T_i$, $v_i \neq \mathbb{G}_{\mathsf{val}T_i}(a)$. By Proposition 8.5, for any environment env, stack $stack$ and store $store$,

$$\begin{aligned} &\langle env, env_{\mathsf{prim}}(\mathbb{G}_{\mathsf{pred}}(T_i))\ v_i, stack, store \rangle \\ \to^* &\langle env', \text{false}, stack, store' \rangle \text{ where } store' \supseteq store \end{aligned}$$

So,

$$\begin{aligned}
Th^o &= \langle env^o, \mathsf{tagfunction}^{O,\tau}[env_1^o, pm_{\text{true}}(Q)]\ v, stack^o, store^o\rangle \\
&\to^* \langle env_2^o, pe_2^o, stack^o, store^o\rangle \\
&\quad \text{where } env_2^o \stackrel{\text{def}}{=} env_1^o[\mathbb{G}_{\mathsf{var}}(x_1) \mapsto v_1, \ldots, \mathbb{G}_{\mathsf{var}}(x_k) \mapsto v_k], \\
&\qquad pe_2^o \stackrel{\text{def}}{=} \mathsf{if}\ (\mathbb{G}_{\mathsf{pred}}(T_1)\ \mathbb{G}_{\mathsf{var}}(x_1))\ \&\&\ \ldots\ \&\&\ (\mathbb{G}_{\mathsf{pred}}(T_k)\ \mathbb{G}_{\mathsf{var}}(x_k)) \\
&\qquad\quad \mathsf{then}\ (\mathbb{G}_{\mathsf{file}}(x_1[\widetilde{i}]); \ldots; \mathbb{G}_{\mathsf{file}}(x_k[\widetilde{i}]); \mathbb{G}(P)) \\
&\qquad\quad \mathsf{else\ raise\ Bad_Call} \\
\to^*\ Th_1^o &\stackrel{\text{def}}{=} \langle env_2^o, \mathsf{raise\ Bad_Call}, stack^o, store_1^o\rangle
\end{aligned}$$

where $store_1^o \supseteq store^o$ by Proposition 8.5 applied k times.

If the oracle O is not under replication, then (FailedCall1) applies, so either $O[\widetilde{a}] \notin \mathcal{I}$ and in this case by Property 6(b)i, $store^o[l_{\mathsf{tok}}(O[\widetilde{a}])] = \text{false}$, or there exists i such that $\forall a \in T_i$, $v_i \neq \mathbb{G}_{\mathsf{val}T_i}(a)$, so we have a reduction similar to the case in which O is under replication.

Let $\mathcal{CT}_1$ be an extension of the trace $\mathcal{CT}$ until $\mathcal{C}[\mathsf{Th} \mapsto Th_1^o]$ and $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - 1, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}}$. Let us prove that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. The current expression is an exceptional value, so replacecalls allows any environment in the current thread, and $store_1^o \supseteq store^o$, so Property 6(b)i is preserved for the current thread. The OCaml side uses more reductions than the simulator side, so Property 15 is preserved. There is one more oracle call, and α and $\mathcal{I}$ are unchanged, so Properties 16 and 17 are preserved. The other properties are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$.

Case 2.8. Let us finally deal with the remaining cases. Cases 2.1, 2.2, 2.3, 2.4 present all cases in which $\mathcal{CS}$ does not reduce. Case 2.6 covers the reduction rule (Simulator add thread). So $\mathcal{CS}$ reduces using rule (Simulator toplevel). So let us denote $\mathcal{CS}_1$ the configuration such that $\mathcal{CS} = \mathcal{C}^s, \mathcal{RI}, \mathcal{I} \to \mathcal{CS}_1 = \mathcal{C}_1^s, \mathcal{RI}, \mathcal{I}$. Since the case of failed oracle calls is already handled in Case 2.7, $\mathcal{C}^s \to \mathcal{C}_1^s$ is obtained by rules of the OCaml semantics, not by (FailedCall1) or (FailedCall2).

If $pe^s = \mathsf{schedule}(tj')$, then by Property 6(b)i, $pe^o = \mathsf{schedule}(tj')$, so $\mathcal{C}^s$ and $\mathcal{C}$ reduce in the same way using rules (Toplevel schedule1) or (Toplevel schedule2) for $\mathcal{C}^s$ and (New toplevel schedule1) or (New toplevel schedule2) for $\mathcal{C}$. Let $\mathcal{C}_1$ be the configuration such that $\mathcal{C} \to \mathcal{C}_1$ and $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_1$. Let $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - 1, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}}$ and $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$.

In all other cases, $\mathcal{C}^s$ reduces by (Toplevel). By Property 6(b)i, the current thread of the OCaml configuration has the same form as in the simulator configuration: the semantic rules are parametric in the elements that are replaced by replaceinitpm and replacecalls, so the OCaml configuration $\mathcal{C}$ reduces by (New toplevel), using a reduction $Th, globalstore \longrightarrow_p Th', globalstore'$ obtained by exactly the same semantic rules as on the simulator side. Let $\mathcal{C}_1$ be the configuration such that $\mathcal{C} \to \mathcal{C}_1$ and $\mathcal{CT}_1$ be the extension of the trace $\mathcal{CT}$ until $\mathcal{C}_1$. Let $\mathfrak{C}_1^{\mathsf{cs}} \stackrel{\text{def}}{=} E, P, \mathcal{T}, \mathcal{Q}, \mathcal{R}, \mathcal{E}, steps - 1, \mathcal{CS}_1$. We have $\mathfrak{C}^{\mathsf{cs}} \rightsquigarrow \mathfrak{C}_1^{\mathsf{cs}}$ and we briefly show that $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$.

If the reduction touches a local store location l, then by Properties 6(b)i and 6(b)ii, l cannot be in the image of l_{tok} or $l_{\mathsf{init\text{-}tok}}$. Moreover, in all cases, the

reduction commutes with replaceinitpm and replacecalls, so Property 6 holds for $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{CT}_1$. (Since calls to tagged closures are already handled in Case 2.5, we do not consider this case here. This is important, because the reduction would not commute with replaceinitpm in this case: replaceinitpm replaces the pattern-matching inside the tagged closure before the call, but would not replace it in the reduced configuration.) If the reduction touches the global store, that is, it uses rule (Globalstore2), let l be the concerned location; by Property 7, the location l is not in S_{priv}, and in OCaml the same operation is carried out on l. So in all cases, Properties 7, 8, 9, and 10 hold for $\mathfrak{C}_1^{\mathsf{cs}}$ and $\mathcal{CT}_1$. The oracle sets may only decrease, in case a subexpression is removed by reduction, so Properties 13 and 14 are preserved. The reduction is performed in one step on both sides, so Property 15 is preserved. The other properties are inherited from $\mathfrak{C}^{\mathsf{cs}} \equiv \mathcal{CT}$, so $\mathfrak{C}_1^{\mathsf{cs}} \equiv \mathcal{CT}_1$. □